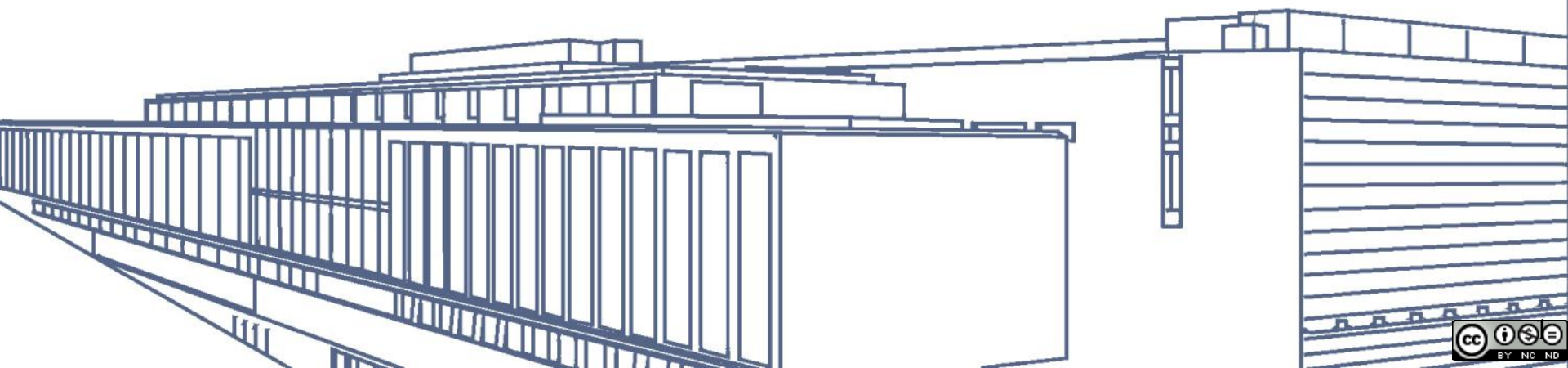


Knowledge Representation and Engineering

Notes



Contents

- 1. Introduction and Concepts**
 - 1.1. Data, Information, and Knowledge
 - 1.2. Types and Uses of Knowledge
 - 1.3. Knowledge Representation
 - 1.4. Knowledge Engineering
 - 1.5. Syntax and Semantics
- 2. Knowledge Representation**
 - 2.1. First Order Logic
 - 2.2. Rules and Production Systems
 - 2.3. Object Oriented Representation
 - 2.4. Network Representation
 - 2.5. Ontologies
- 3. Knowledge Engineering**
 - 3.1. Knowledge Life Cycle
 - 3.2. Knowledge Auditing
 - 3.3. Knowledge Deployment
 - 3.4. Knowledge Acquisition
- 4. Knowledge Representation in the Web**

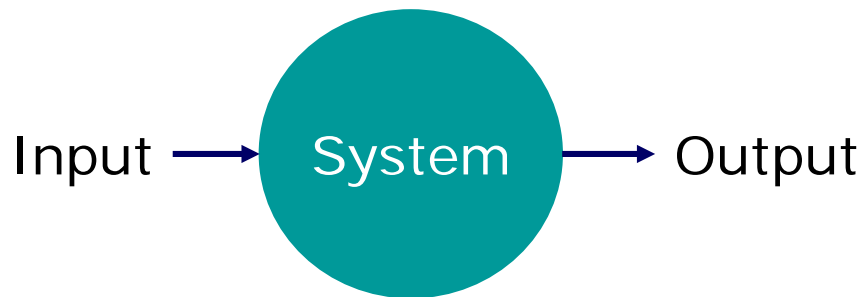
1. Introduction and Concepts

Definition (Knowledge Representation): Knowledge Representation is the area of Artificial Intelligence (AI) concerned with *how knowledge can be represented* symbolically and manipulated in an automated way.

Definition (Knowledge Engineering): Knowledge Engineering is the area of Computer Engineering (CE) concerned with the procedures and *methods that help developers to systematically and formally construct knowledge bases*.

Conceptualization of knowledge-based systems

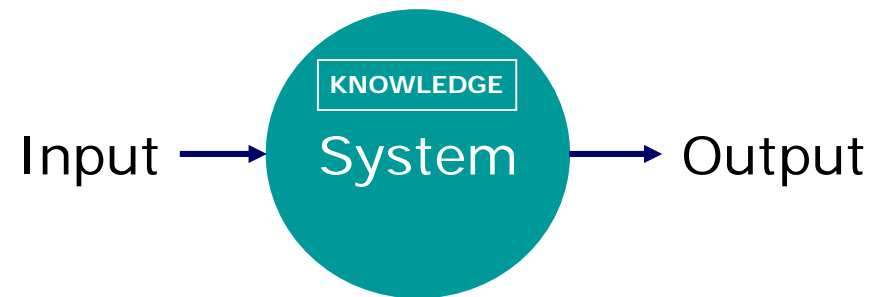
mechanical behaviour



$$\text{Output} = f(\text{Input})$$

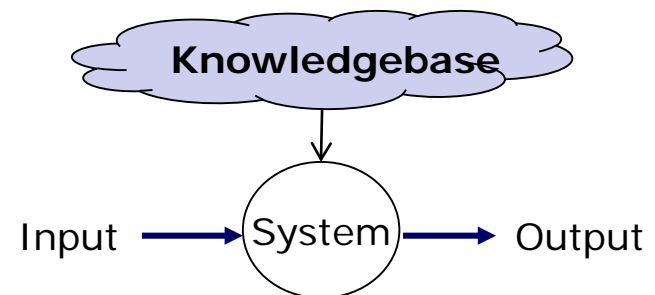
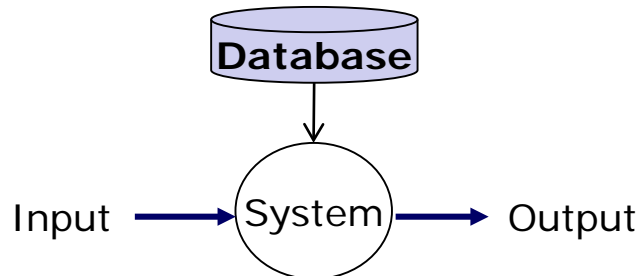
Ex. arithmetic operations
2 times 2 is always 4

intelligent behaviour



$$\text{Output} = f(\text{Input}, \text{KNOWLEDGE})$$

Ex. deciding the best drug
symptoms ⊕ medical knowledge ⇒ drug



Data is not enough

- Data Bases

PERSONS	
Id	Name
A	Albert
B	Beth
C	Cindy
...	

PARENTSHIPS	
X	Y
A	B
B	C
B	D
...	

- Types of questions that we can answer:

- Parent (A, B)?

- Parent (x, B)?

```
SELECT X FROM PARENTSHIPS WHERE Y="B" ;
```

- Parent (A, y)?

```
SELECT Y FROM PARENTSHIPS WHERE X="A" ;
```

- What if we want to know about '*ancestors*'?

- A new table ANCESTORS(X,Y) is required

- This table is huge !!!!

How huge?

- People: n
- *Ancestorships*: $O(n^2)$
- Ex. If the average number of siblings in the World is 2.36, for one person:
 - in 1 generation, we have 2.36 ancestorships
 - in 2 generations, 5.57
 - in 3 generations 16.71
 - in 10 generations 5,359, and
 - in 20 more than 28.7 million ancestorships to store.



ancestor

Knowledge as Knowledge Bases

- Alternatively, we can provide the computer with “knowledge” (i.e., intelligence). So, we can make the computer to know that:

ANCESTOR

Parent (X, Y) => Ancestor (X, Y)

Ancestor (X, Z) & Parent (Z, Y) => Ancestor (X, Y)

- This is explicit knowledge in the Knowledge Base
- Implicit data to be calculated with the explicit knowledge
- New questions we are able to answer:
 - Ancestor (X, Y) ?

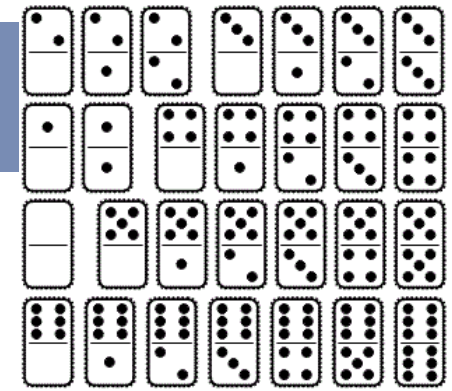
```
If PARENT(X, Y) then return true;
else
  search Z: PARENT (Z, Y)
  if such Z does not exist then return false;
  else {only two possible parents}
      Ancestor(X,Z.father) or Ancestor(X,Z.mother);
```

Knowledge Domains

- Knowledge is convenient in many domains:
 - Building Houses
 - Stock Markets
 - Chemical Processes inside Bacteria
 - Medicine (diagnosis, treatment, prognosis)
 - Olympic Games
 - Games (chess, domino, etc.)
 - Networking and Social Networks
 - ...

- Decision Support Systems: computer systems that help users to make decisions in complex intellectual settings.

Examples

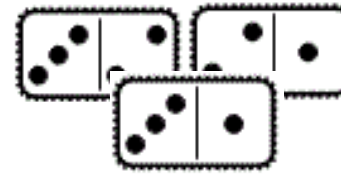


■ Domino: detect correct plays

– Dominoes: $[0,0]$, $[0,1]$, $[1,0]$, $[0,2]$, ...

– Knowledge:

$$[X,Z] \ \& \ [Z,Y] \Rightarrow [X,Y]$$



– Alternatively:

$$\text{Free}(x,z) \ \& \ \text{Ontable}(z,y) \Rightarrow \text{Ontable}(x,y) \ \& \ \text{not}(\text{Free}(x,z))$$

$$\text{Ontable}(x,y) \Rightarrow \text{Ontable}(y,x)$$

■ Trigonometry: check for correct areas

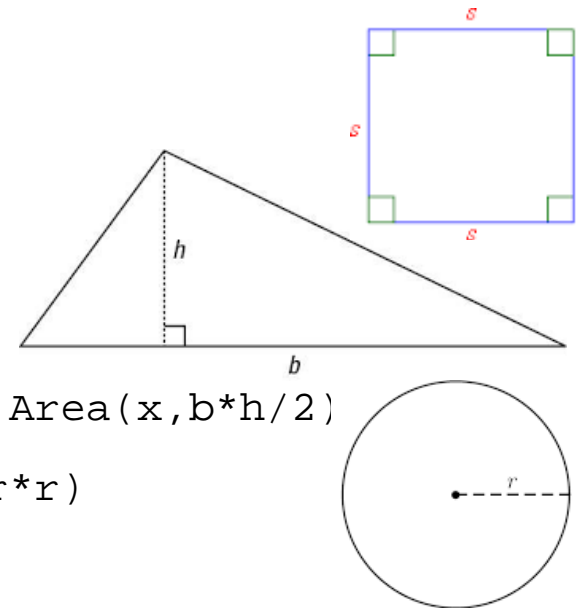
– Knowledge:

$$\text{Square}(x) \ \& \ \text{Side}(x,s) \Rightarrow \text{Area}(x,s*s)$$

$$\text{Triangle}(x) \ \& \ \text{Base}(x,b) \ \& \ \text{Height}(x,h) \Rightarrow \text{Area}(x,b*h/2)$$

$$\text{Circle}(x) \ \& \ \text{Radius}(x,r) \Rightarrow \text{Area}(x,3.1416*r*r)$$

...



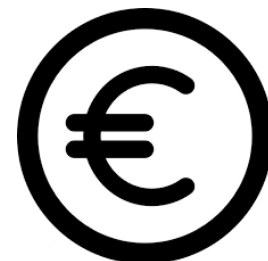
... and more

■ Stockmarket

- Shares: a, b, c, \dots
- Knowledge:

$Risk(X, R) \ \& \ ExpectedBenefits(X, B) \ \& \ B < R \Rightarrow Sell(X)$
 $Falling(X, F) \ \& \ F > 15\% \ \& \ Affects(X, Y) \Rightarrow Sell(Y)$

...



■ Allergies

- Substances: a, b, c, \dots
- Knowledge

$Substance(X) \ \& \ AllergyTo(X) \ \& \ Has(Y, X) \Rightarrow AllergyTo(Y)$
 $Has(X, Y) \ \& \ Has(Y, Z) \Rightarrow Has(X, Z)$

...

■

...



1.1. Data, Information and Knowledge

- Data, Information, and Knowledge are three related terms, ...
... but they mean different things in KRE.



Data



- A) set of discrete, objective facts about events. Data is transformed into information by adding value through context, categorisation, calculations, corrections, and condensation.
- B) facts and figures, without context and interpretation.
- The nature of data is raw and without context. It simply exists and has no significance beyond its existence and value. It can exist in any form, usable or not.

Single value: **190** (kg, km, population, exams, sons, ... ?)

Multiple value: (**green, ugly, biped, grumpy**) (bug, reptile, Martian, ...)

Numerical / Categorical OR Quantitative / Qualitative

Information



- A) a message, usually in the form of a document or an audible or visible communication meant to **change the way the receiver perceives something**, to have an impact on his judgement and behaviour.
- B) patterns in the data.
- Information is data that have been given a meaning by way of context.



Single value: 190 kg.

Multiple value: (green, ugly, biped, grumpy)

Knowledge



- A fluid mix of framed experience, values, contextual information, and expert insight that provides a framework for evaluating and incorporating new experiences and information. It originates and is applied in the minds of “knowers”. In organisations, it often becomes embedded not only in documents or repositories but also in organisational routines, processes, practices, and norms.
- Actionable information.
- The integration of ideas, experience, intuition, skill, and lessons learned that has the potential to create value for a business, its employees, products and services, customers and ultimately shareholders by informing decisions and improving actions.
- Knowledge is information combined with understanding and capability; it “lives” in the minds of people. Typically, knowledge provides a level of predictability that usually stems from the recognition of patterns.
- Knowledge is information that has been generalized to increase applicability.



Data + “*meaning*” = Information

- Sorts of “*meanings*” (*the five C’s by Davenport&Prusak*):
 - Contextualization: the purpose of the data gives a meaning.
(*ex. Clients that will be emailed*)
 - Categorization: the data are classified / generalized in concepts.
(*ex. Company clients vs. Autonomous clients*)
 - Calculation: the meaning is given by a mathematical or statistical analysis.
(*ex. Good client = buys \geq \$1 million*)
 - Correction: the meaning comes from the removal of errors from the data.
(*ex. Expenditure in £ (instead of €) inform about English clients*)
 - Condensation: data is summarized in a more concise form, without unnecessary elements.
(*ex. Incentives out of client data gives info about enterprise incentives*)

john@fakemail.com
mary@fakemail.com
peter@fakemail.com
sophie@fakemail.com
charles@fakemail.com
eve@fakemail.com
george@fakemail.com
sylvia@fakemail.com

John
 Mary
 Peter
 Sophie
 IBM
 Oracle
 Microsoft
 SAP

John €23000
 Mary €539
 Peter €1023000
 Sophie €234562
 IBM €3456973
 Oracle €1003496
 Microsoft €573045
 SAP €478328

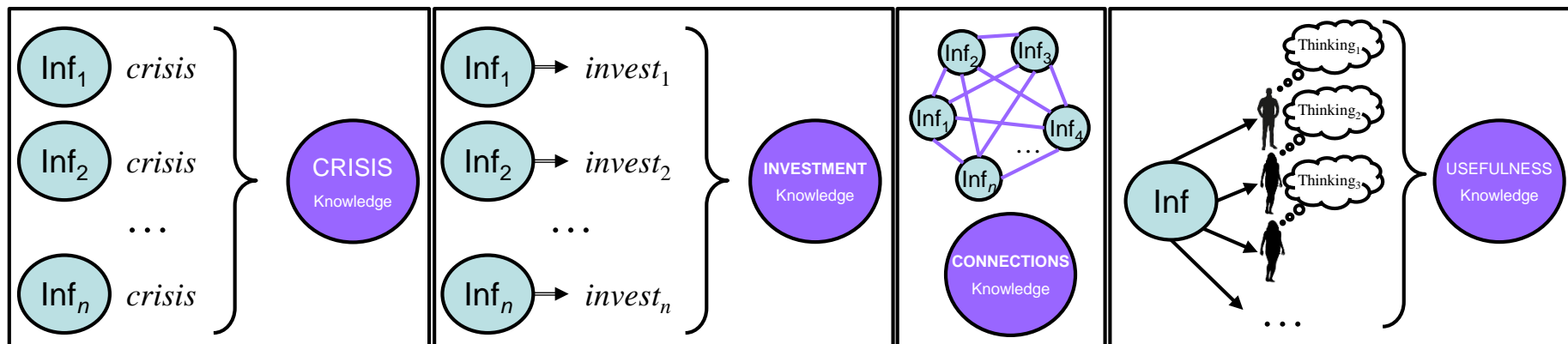
€23000
 €1023000
 €234562
 \$3456973
 €1003496
 \$573045
 €478328
 TOTAL ERROR

Id	Nationality	Gender	Salary	Complement
1	Spanish	M	6000	1200
2	French	F	4000	875
3	Spanish	M	7500	300
...				
	Nationality	Gender	Salary	Complement
	Spanish	M	3574.6	834.6
	Spanish	F	2906.9	945.1
	...			

Condensed

Information + “something” = Knowledge

- Sorts of “something” (the four C’s by Davenport&Prusak):
 - Comparison: is this information representing something similar to other situations?
(ex. Defining a firm crisis)
 - Consequences: implications of the information in company decisions and actions.
(ex. Identify moments in which the firm invested)
 - Connections: how the information is related to other information.
(ex. There is a ratio $\sim 2/1$ between incomes and investment)
 - Conversations: what people think about some information.
(ex. Useful / useless concepts)
- *Something* = application (Tobin, 1998)



Knowledge + *intuition* + *experience* = Wisdom

- Other upper to Knowledge concepts:

- Wisdom:

knowledge + intuition + experience

- Expertise (*perícia* in Spanish):

wisdom + selection + principles + constrains + learning

- Capability:

expertise + integration + distribution + navigation

1.2. Types and Uses of Knowledge

- According to the targeted feature we can classify knowledge in different **types** (sorts of knowledge) and applicable to different **uses** (uses of knowledge).

Sorts of Knowledge (i): evidence

- Explicit Knowledge: the kind of knowledge which can be expressed in words and numbers and shared in the form of data, scientific formulae, product specifications, manuals, universal principles, etc. This kind of knowledge can be transmitted across individuals formally and systematically. It can be processed by a computer, transmitted electronically, or stored in databases.
- Implicit or Tacit Knowledge: the kind of knowledge which can be found in the heads of employees, the experience of customers and the memories of past vendors. It is highly experiential, difficult to document in any detail, ephemeral and transitory.



Sorts of Knowledge (ii): purpose

- Declarative Knowledge or *know-what*: factual assertions an organisation makes about itself, its capabilities, and the marketplace. With this knowledge you know what are the tasks that you have to do.
- Procedural Knowledge or *know-how*: business and organisational processes and strategies of the company. With this knowledge you know how you are supposed to do the tasks that you have to do.

We do what we do because of our *know-what*

We do what we do the way we do it because of our *know-how*

Sorts of Knowledge (iii): ownership

- Individual Knowledge: personal skills, expertise, and experience of each employee of a company about the company processes and the company related domains.
- Group Knowledge: understanding of company groups of employees (i.e. collectives) as they collaborate and co-operate. This includes all the individual knowledge of each of the employees in the group and some extra added value.
- Organizational Knowledge: knowledge held by the organization as a whole.



Sorts of Knowledge (iv): format

- Informal Knowledge: natural language oral, textual or graphical representation of the knowledge (ex. *.TXT).
- Semi-Structured Knowledge: informal representation of knowledge enriched with some attributes (ex. *.XML).
- Structured Knowledge: the knowledge is represented according to some attribute-based structures (ex. *.DB2)
- Formal Knowledge: the knowledge is represented by means of knowledge structures as frames, production rules, ontologies, etc.

"You know? Mary got married!"

informal

```
<person name="Mary">
  <married>Yes</married>
  ...
</person>
```

semi-structured

Name	Married
Mary	true
...	...

structured

```
Person(Mary);
Married(Mary);
...
```

formal

Knowledge Uses in an Enterprise

	<i>explicit</i>	<i>implicit</i>	<i>know-what</i>	<i>know-how</i>
Business Strategies	Y	N	N	Y
Products and Services	Y	N	Y	N
Business Processes	N	Y	N	Y
Organisational Structures	Y	N	Y	N
Policies and Procedures	Y	Y	Y	Y
Culture and Values	Y	Y	Y	N
Information Systems	Y	Y	Y	N

Business strategy: “our approach to clients is first by a mailing campaign during one month then direct contact”

Products and services: “all of our products are made with first class raw materials, and never plastic”.

Business processes: procedures followed by the people of the commercial department.

Organisational Structures: “we have three departments: production, accounting, and marketing”

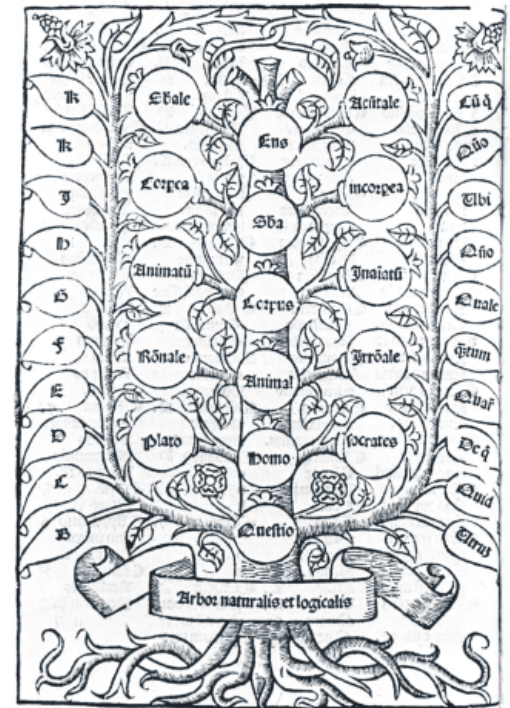
Policies and procedures: “our company has a gender parity policy for hiring people”

Culture and values: “our company promotes good relationship between our workers”

Information systems: “every time a sell is confirmed, you have to fill in a template with the data about the sell”

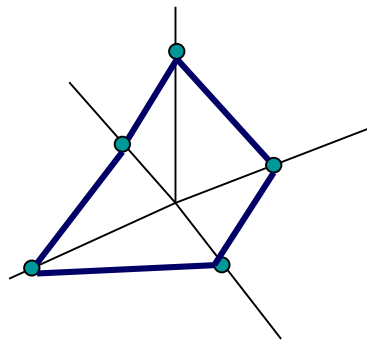
1.3. Knowledge Representation

- Short review of...
 - ... data representation and organization
 - ... information representation
- Can data & information structures be used to represent knowledge?
- Knowledge Representation summary and needs.
- Alternative Knowledge Representation formalisms:
 - *First Order Logic*
 - *Rules and Production Systems*
 - *Object Oriented Representation*
 - *Network Representation*
 - *Ontologies*



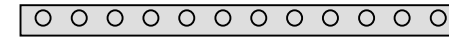
Data representation and organization

- Single value representation
- Multiple value representation
 - Data structures
 - Vector, array, list, etc.
 - Matrix.



$D(x,y,z, \dots)$

○



$D(x)$



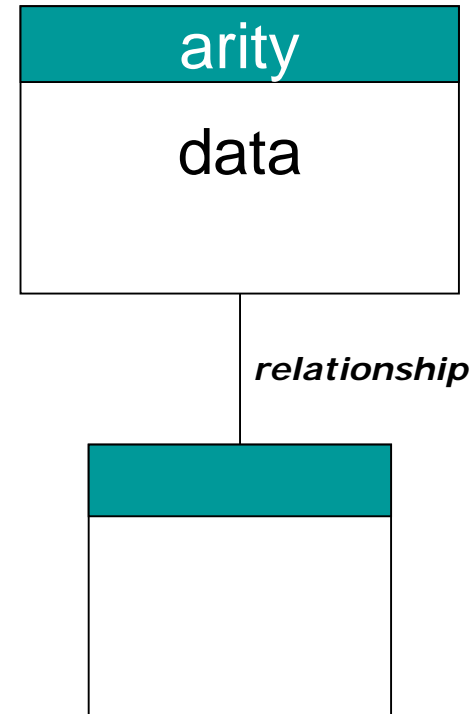
$D(x,y)$



$D(x,y,z)$

Information representation

- Information Systems
- Table
 - Column Heading = typed feature
 - Row = instance
 - Cell = (single) data
 - *Intension vs. Extension*
- Data bases
 - Relationship: column to column
 - Cardinality: 1, N
 - Optionality: 0 allowed? Y/N
- Data warehouses



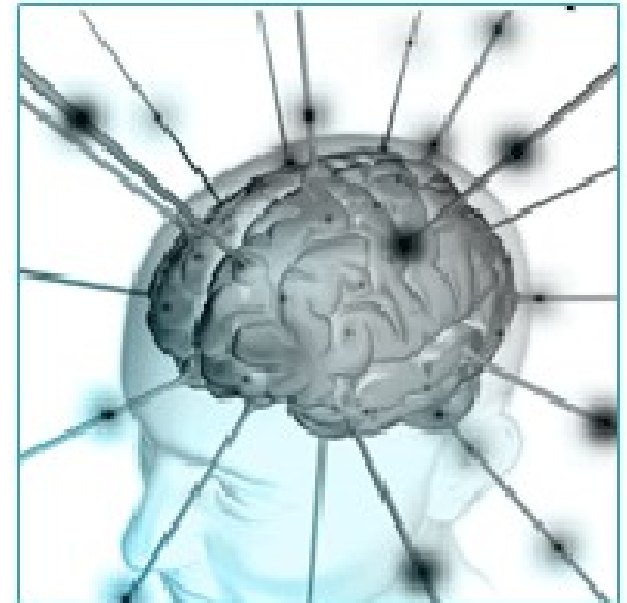
Knowledge representation and modelling

KR aims at expressing knowledge in a computer manageable way, so that it can be used in an computer intelligence process.

- KR issues:
 - Syntax: structures that support the representation.
 - Semantics: meaning of the knowledge represented.
 - Reasoning & Inference: process by which knowledge is used to obtain conclusions.
- Inference alternatives:
 - Forward chaining (*modus ponens*): $A, A \rightarrow B \vdash B$
 - Backward chaining (*modus tollens*): $\neg B, A \rightarrow B \vdash \neg A$
- Knowledge-base issues:
 - Completeness: given a KB, the inference process can find B or $\neg B$, for any correct assertion B.
 - Soundness: given a KB, the inference process cannot find both B and $\neg B$, for any correct assertion B.

Knowledge Representation Formal Models

- *First Order Logic*
- *Rules and Production Systems*
- *Object Oriented Representation*
- *Network Representation*
- *Ontologies*



1.4. Knowledge Engineering

- Software engineering (SE): methods to help developing software
- Intelligent systems are a concrete sort of software. So,

Can SE be used for knowledge engineering?

- Knowledge Engineering summary and needs.
- Alternative Knowledge Engineering Methods

Software Engineering

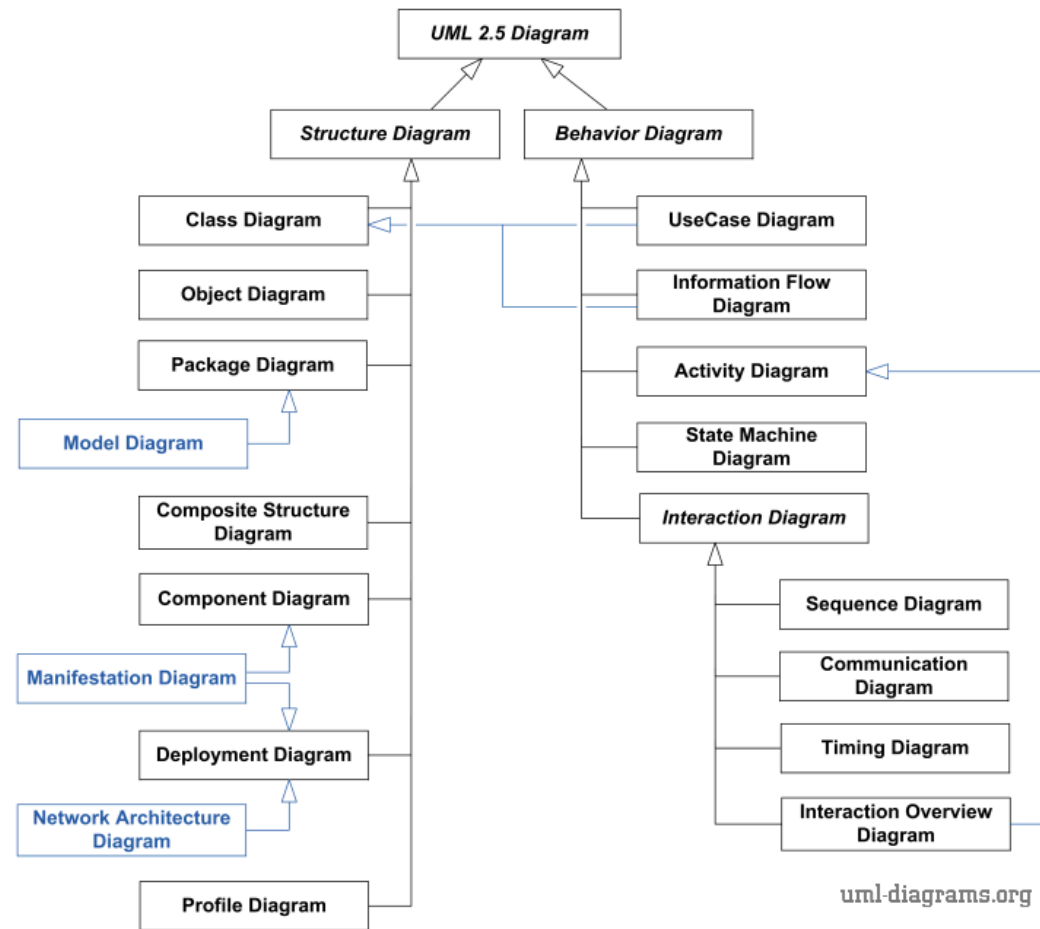
UML 2.5 approach

Structure diagrams: they show the *static structure* of the system and its *parts* on different abstraction and implementation levels and how they are related to each other. The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

Behavior Diagrams: they show the *dynamic behavior* of the objects in a system, which can be described as a series of changes to the system over time.

SE basic tasks:

- Specification of Requirements
- Analysis of the Problem
- Design of the System
- Implementation
- Testing
- Installation
- Maintenance



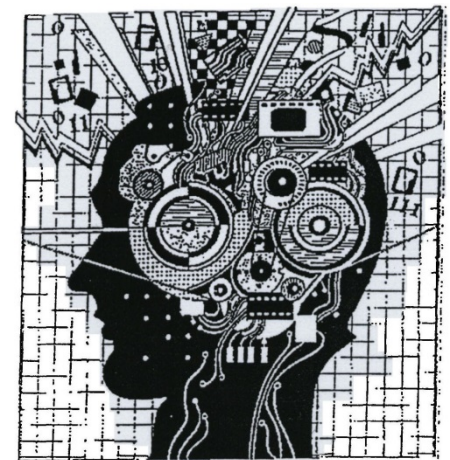
<http://www.uml-diagrams.org/uml-25-diagrams.html>

uml-diagrams.org

Knowledge Engineering

KE aims at capturing, storing, and validating knowledge in a computer, so that it can be used in an computer intelligence process.

- KE issues:
 - Knowledge Life Cycle: knowledge processes, uses, and loops.
 - Knowledge Auditing: knowledge delimitation.
 - Knowledge Deployment: installing knowledge.
 - Knowledge Acquisition (or Knowledge Elicitation): knowledge expert and knowledge engineer cooperation.



Knowledge Engineering Formal Methods

- Some KE methods are:

- *Common KADS*

- AT Schreiber, JM Akkermans, et al.: Knowledge Engineering and Management. The CommonKADS Methodology.

- *MIKE: Model-based and Incremental Knowledge Engineering*

- J Angele, D Fensel, D Landes, R Studer: Developing Knowledge-Based Systems with MIKE.

- *101*

- NF Noy, DL McGuinness: Ontology Development 101: A Guide to Creating Your First Ontology

1.5. Syntax and Semantics

- *Syntax*: the way that we express knowledge.
- *Semantics*: the meaning of these expressions.
- Case 1:
 - Syntax: “*for all x , flies(x)*”. (it is syntactically correct, but ...)
 - Semantics: It’s true in the world of birds, but not in the world of animals.
- Case 2:
 - Syntax: “*for all x , flies(x) \supset feathered(x)*”.
 - Semantics: It can be true in the world of birds, but not in the world of dinosaurs (or insects, bats, ...).
- One knowledge representation (*syntax*) can be useful in many worlds (*semantics*), but useless in many other worlds.

Conclusions to Chapter 1

- I know the singularity of knowledge-based systems
- I know how to distinguish between **data, information, and knowledge**.
- I know the **processes to transform data into information**.
- I know the **processes to bring information into knowledge**.
- I know **how to isolate knowledge** from textual descriptions.
- I know about different **sorts of knowledge**:
 - implicit vs. tacit
 - declarative vs. procedural
 - individual vs. group vs. organizational
 - informal vs. semi-structured vs. structured vs. formal
- I know how to **identify declarative and procedural knowledge**.
- I know multiple **uses of knowledge** in an enterprise, organization, domain, etc.
- I know how to justify why **knowledge representation languages** are needed.
- I know how to justify why **knowledge engineering methods** are needed.
- I know the broad concepts of Knowledge Representation and Knowledge Engineering.
- I know about the importance of **Syntax and Semantics** in knowledge-based systems.

2. Knowledge Representation

- **Definition** (***Knowledge Representation***): Knowledge Representation (and Reasoning) is that part of Artificial Intelligence that is concerned with how an agent describes what it knows, and uses it in deciding what to do.
- The most used formalisms to represent knowledge are:
 - ***First Order Logic***
 - ***Rules and Production Systems***
 - ***Object Oriented Representation***
 - ***Network Representation***
 - ***Ontologies***

2.1. First Order Logic (FOL)

■ FOL Symbols

- *logical symbols*: fixed meaning or use (like constants). There are three sorts of logical symbols:
 - punctuation: parentheses $(,)$, and point $(.)$
 - connectives: $\wedge, \vee, \neg, \exists, \forall, =, \supset$ (also \Rightarrow)
 - variables: x, y, z, \dots
- *non-logical symbols*: application-dependent meaning or use. There are two sorts of non-logical symbols:
 - function symbols: f, g, h, \dots Ex. *bestFriend*(x) as a function calculation.
 - predicate symbols: P, Q, R, \dots Ex. *BestFriend*(*John*,*Mike*) as a statement.
- **Definition** (*Arity*) : Arity is the number of arguments of non-logical symbols. Ex. *Dog*(*Tobby*), *OlderThan*(*Tom*,*Jerry*), *Parents*(*Joseph*, *Mary*, *Jesus*), ... (arity 0 for constants *High*())

Syntactic Expressions in FOL

- There are two sort of syntactic expressions in FOL:
 - *Terms*:
 - (i) every variable is a *term*
 - (ii) t_1, \dots, t_n *terms* and f a function, then $f(t_1, \dots, t_n)$ is a *term*

Ex. 2 and 5 terms, addition(x, y) a function, then addition(2,5) is a term (expectedly equivalent to 7, but this depends on how the function addition is defined).
 - *Formulas*:
 - (i) if t_1, \dots, t_n are *terms* and P is a predicate symbol, then $P(t_1, \dots, t_n)$ is a *formula*
 - (ii) if t_1 and t_2 are *terms*, then $t_1 = t_2$ is a *formula*
 - (iii) if a, b are *formulas* and x a variable, then $\neg a, a \vee b, a \wedge b, \forall x.a, \exists x.b$ are *formulas*.

Ex. $\forall x. \exists y. (\text{person}(x) \supset (\text{person}(y) \wedge \text{mother}(y, x)))$ is a formula representing the fact that “all the persons have a mother (which is also a person)”

Note that \wedge has preference over \vee . Ex. $a \vee b \wedge c$ is equivalent to $a \vee (b \wedge c)$.
Note that \neg has preference over \wedge and \vee . Ex. $\neg a \vee b$ is equivalent to $(\neg a) \vee b$.

Implication, equivalence and sentences in FOL

- **Definition** (*Implication*): $a \supset b$ stands for $(a \Rightarrow b)$, or $\neg a \vee b$.

Ex. $\text{Man}(x) \supset \text{Human}(x)$.

- **Definition** (*Equivalence*): $a \equiv b$ stands for $(a \supset b) \wedge (b \supset a)$

Ex. $\text{Man}(x) \equiv \text{Human}(x) \wedge \text{Male}(x)$.

- Bounded&Free Variables:

$\forall x. \text{Relative}(x, \text{John})$.

$\forall x. \text{Relative}(x, y)$. – x is a bounded variable, y is a free variable

- **Definition** (*Sentence*): In FOL, a sentence is any formula without free variables.

Some Operational Equivalences

1. $x \equiv x$
2. $x \equiv \neg(\neg x)$
3. $(x \wedge y) \equiv \neg((\neg x) \vee (\neg y))$
4. $(x \vee y) \equiv \neg((\neg x) \wedge (\neg y))$
5. $(x \supset y) \equiv (\neg x) \vee y$
6. $\forall x: P(x) \equiv \neg(\exists x: \neg P(x))$
7. $\exists x: P(x) \equiv \neg(\forall x: \neg P(x))$

Knowledge representation with FOL

- Represent explicit knowledge as facts (ex. $\text{Dog}(\text{Tobby})$) or sentences (ex. $\forall x. \text{Dog}(x) \supset \text{Mammal}(x)$).
- Do not represent implicit knowledge (ex. $\text{Mammal}(\text{Tobby})$).
- Representing know-what knowledge:

- **Object properties:**

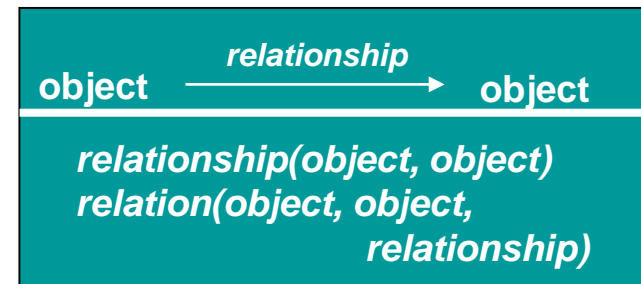
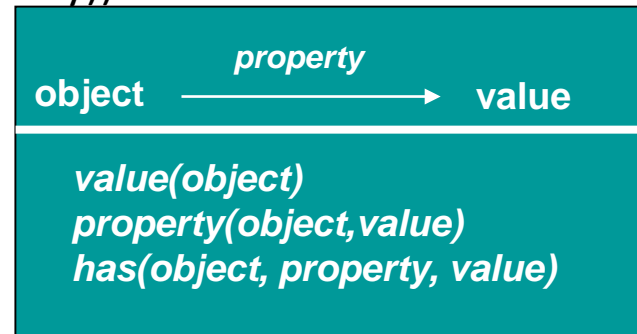
Ex. “The Sun is yellow”, “A person has two legs”, etc.

- Alternative representations:
 - *Values as predicates* $\text{value}(\text{object})$:
ex. $\text{Yellow}(\text{sun})$, $\text{TwoLeg}(\text{person})$, etc.
 - *Properties as predicates* $\text{property}(\text{object}, \text{value})$:
ex. $\text{Color}(\text{sun}, \text{yellow})$, $\text{Legs}(\text{person}, 2)$, etc.
 - *Complex predicates* $\text{p}(\text{object}, \text{property}, \text{value})$: has, etc.
ex. $\text{Is}(\text{Sun}, \text{color}, \text{yellow})$, $\text{Has}(\text{person}, \text{leg}, 2)$, etc.

- **Object relationships:**

Ex. “Joseph and Mary are married”, “Jesus is the son of Joseph and Mary”, etc.

- Alternative representations:
 - *Relationship as predicates*
ex. $\text{Married}(\text{joseph}, \text{mary})$, $\text{ParentsOf}(\text{joseph}, \text{mary}, \text{jesus})$, etc.
 - *Complex predicates*:
ex. $\text{Relation}(\text{joseph}, \text{mary}, \text{marriage})$, $\text{Relativeness}(\text{joseph}, \text{mary}, \text{jesus}, \text{parentship})$, etc.



Representing n-Ary Relationships

$(\text{object}_1, \dots, \text{object}_{i-1}, \underline{\text{object}_i}, \text{object}_{i+1}, \dots, \text{object}_n).$

1. $\text{Object}_i(\text{object}_1, \dots, \text{object}_{i-1}, \text{object}_{i+1}, \dots, \text{object}_n).$

Ex. "John, Mary, and Lou live together with Sally": $\text{LiveWithSally}(\text{John}, \text{Mary}, \text{Lou}).$

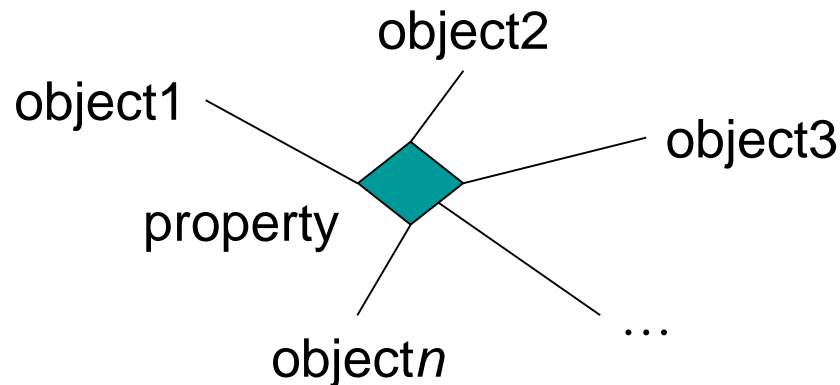
2. $\text{Property}(\text{object}_1, \text{object}_2, \text{object}_3, \dots, \text{object}_n).$

Ex. "Cats eat birds sometimes": $\text{Eats}(\text{Cat}, \text{Birds}, \text{Sometimes}).$

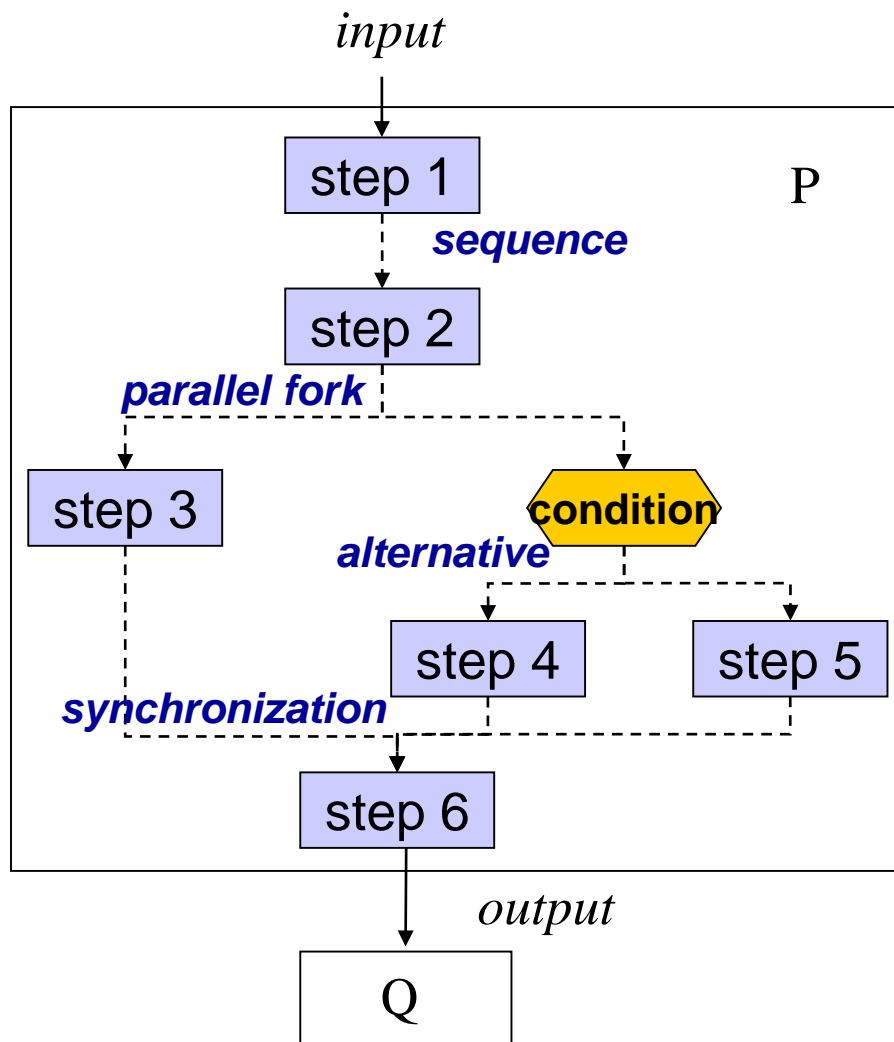
Ex. "John purchases a cat from a pet-shop for \$200 as a birth gift": $\text{purchases}(\text{John}, \text{Cat}, \text{Pet-Shop}, \$200, \text{birth-gift}).$

3. $\text{Has/Are}(\text{object}_1, \dots, \text{object}_{i-1}, \text{object}_{i+1}, \dots, \text{object}_n, \text{property}).$

Ex. "Bird feathers are clean and shining": $\text{Has}(\text{Bird}, \text{Feathers}, \text{Clean}, \text{Shining}).$



Representing know-how knowledge



- procedural control models:
sequence, parallel fork, synchronization, alternatives, ...
- non-monotonic logic
rules may reduce the KB
(ex. which steps are the next ones to be applied)
- The knowledge base:

Start(P,input)

--control knowledge

$\forall x. \text{Start}(P,x) \supset \text{Started}(s1, x, 0).$
 $\forall x. \forall i. \text{Started}(s1, x, i) \supset \text{Started}(s2, \text{step1}(x), i+1).$
 $\forall x. \forall i. \text{Started}(s2, x, i) \supset \text{Started}(s3, y, i+1) \wedge \text{Started}(c, y, i+1) \wedge y = \text{step2}(x).$
 $\forall x. \forall i. (\text{Started}(c, x, i) \wedge \underline{C}(x, i)) \supset \text{Started}(s4, x, i+1).$
 $\forall x. \forall i. (\text{Started}(c, x, i) \wedge \neg C(x, i)) \supset \text{Started}(s5, x, i+1).$
 $\forall x. \forall y. \forall i. \forall j. (\text{Started}(s3, x, i) \wedge \text{Started}(s4, y, j) \supset \text{Started}(s6, w, \max(i, j)+1) \wedge w = \text{combine}(\text{step3}(x), \text{step4}(y)).$
 $\forall x. \forall y. \forall i. \forall j. (\text{Started}(s3, x, i) \wedge \text{Started}(s5, y, j) \supset \text{Started}(s6, w, \max(i, j)+1) \wedge w = \text{combine}(\text{step3}(x), \text{step5}(y)).$
 $\forall x. \forall i. \text{Started}(s6, x, i) \supset \text{Start}(Q, \text{step6}(x)).$

Three predicates: Start, Started, **C**
+ Three functions: *combine*, *max*, +

Example 1: colouring

colour(snow, white).

colour(sky, blue).

colour(cloud, white).

...

$\forall x. \forall y. \forall c. \text{madeOf}(x, y) \wedge \text{colour}(y, c) \supset \text{colour}(x, c).$

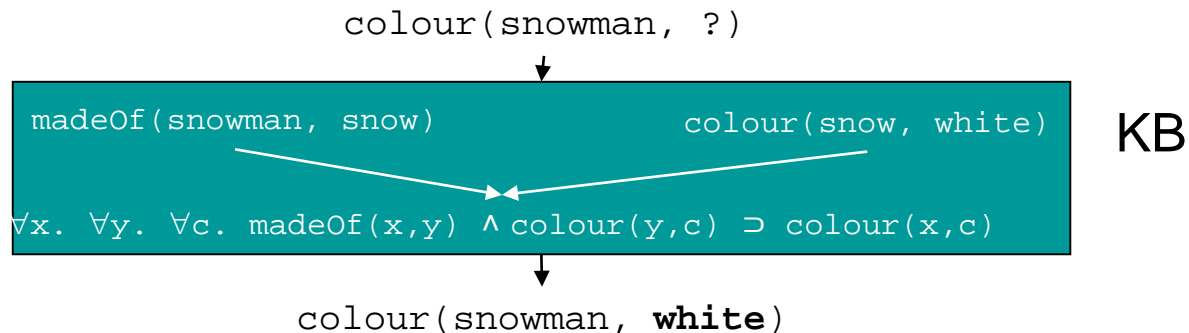
madeOf(grass, vegetation).

colour(vegetation, green).

--you don't need to say that the color of grass is green

madeOf(snowman, snow).

--you don't need to say that the color of snowmen is white



Example 2: world of blocks

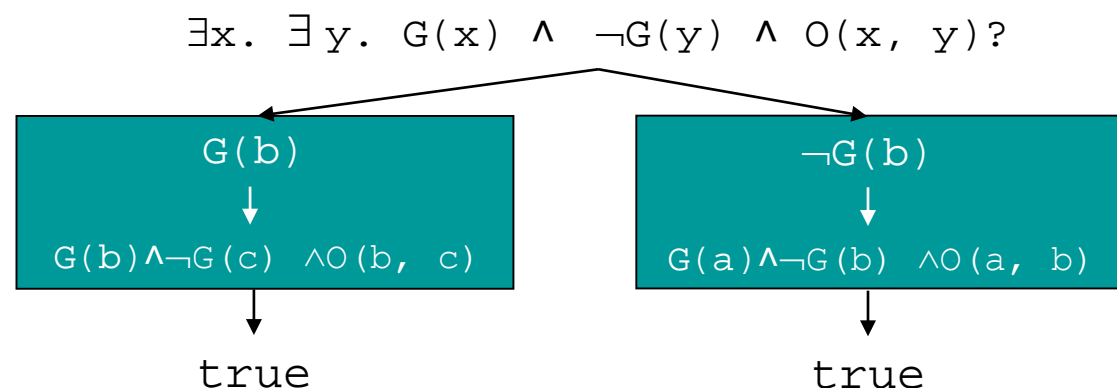
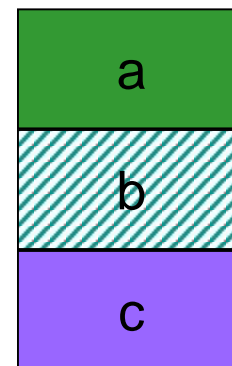
- Three objects: a,b,c
- Two predicate symbols: G (green), O (on)
- Knowledge Base:

$O(a, b).$

$O(b, c).$

$G(a).$

$\neg G(c).$



Constructing FOL Knowledge-Bases (a recipe)

1. Identify the **named individuals** in the domain, either people, animals, objects, places, companies, etc.
Ex. maryJones, johnSmith, urv, toby, etc.
2. Identify **no-named individuals** in the domain.
Ex. maryJones'sLeftEarRing, window5JohnSmith'sHouse, etc.
3. Identify the basic **types** of objects that the individuals are, represented as unary predicates.
Ex. Person(maryJones), University(urv), Dog(toby) Window(window5JohnSmith'sHouse), etc.
4. Identify the set of **attributes** (or **properties**) that the individuals have and that are relevant.
Ex. Rich(johnSmith), Bankrupt(maryJones), etc.
5. Identify the **relationships** between our individuals in our world.
Ex. MarriedTo(maryJones, johnSmith), BlackMails(maryJones, URV), etc.
6. Identify the **functions** of the domain (they must be defined total).
Ex. fatherOf(x), bestFriendOf(x), studiesAt(x), etc.
7. Express **constraints** in the domain (generic or specific)
Ex. Reciprocity of marriage ($\forall x. \forall y. \text{Married}(x,y) \supset \text{Married}(y,x)$)
Ex. All URV workers are professors ($\forall x. \text{WorksAt}(x, \text{urv}) \supset \text{Professor}(x)$)

Example 3: the soap opera world

- Named individuals: john, jane, jim, faultyInsuranceCo, TDiner, ...
- No-named individuals: butcherknife1, ...
- Types: Man, Woman, Company, Restaurant, Knife, ...
- Properties: Rich, HappilyMarried, Loves, WorksFor, Bloody, ...
- Relationships: Loves(x,y), Blackmails(x,y), ...
- Functions: bossOf(x), bestFriendOf(x), ...
- Knowledge Base:
 - Man(john), Woman(jane), Company(faultyInsuranceCo), Restaurant(TDiner), Knife(butcherknife1), ...
 - Rich(john), \neg HappilyMarried(jim), WorksFor(jim, fic), Bloody(butcherknife1), ...
 - Equalities: fic = faultyInsuranceCo, john = bossOf(jim), bestFriendOf(jim) = john, ...
- Constraints:
 - “all rich men in our world love Jane”: $\forall x. (\text{Rich}(x) \wedge \text{Man}(x) \supset \text{Loves}(x, \text{jane}))$.
 - “all women, with possible exception of jane, love john”: $\forall x. (\text{Woman}(x) \wedge x \neq \text{jane} \supset \text{Loves}(x, \text{john}))$.
 - “no one who loves someone blackmails him/her”: $\forall x. \forall y. (\text{Loves}(x, y) \supset \neg \text{Blackmails}(x, y))$.
 - “Jane loves one of John or Jim”: $\text{Loves}(\text{jane}, \text{john}) \vee \text{Loves}(\text{jane}, \text{jim})$.
 - “somebody is blackmailing John”: $\exists x. (\text{Person}(x) \wedge \text{Blackmails}(x, \text{john}))$.
 - “all the marriages are mary-john, jim-sally, ...”: $\forall x. \forall y. (\text{Married}(x, y) \supset (x=\text{mary} \wedge y=\text{john}) \vee (x=\text{jim} \wedge y=\text{sally}) \vee \dots)$.
 - “john, mary, jim, and sally are the only persons”: $\forall x. (\text{Person}(x) \supset (x=\text{mary}) \vee (x=\text{john}) \vee (x=\text{jim}) \vee (x=\text{sally}))$.



$$\forall E \neq EA$$

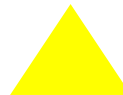
color(x, y) = “the color of x is y”.

■ EA

– $\forall x. \exists y. \text{color}(y, x)$

“All the colors have an object”

(three colors)



– $\exists x. \forall y. \text{color}(y, x)$

“All objects have the same color”



Think of the meaning of: (1) $\forall x. \exists y. \text{color}(x, y)$ and (2) $\exists x. \forall y. \text{color}(x, y)$.

2.2. Rules and Production Systems

- Rule-Based Knowledge Bases have two components:
 - A *Database* of facts
Ex. $\text{Man}(\text{john}), \text{Married}(\text{john}, \text{jane}), \dots$
 - A *Collection* of *rules*
Ex. $\text{Woman}(y) \Leftarrow \text{Man}(x) \wedge \text{Married}(x, y)$
- **Definition** (*Fact*): unit of information that covers the basic truths of the domain
Ex. $\text{Parent}(\text{john}, \text{peter})$.
- **Definition** (*Rule*): universally quantified (\forall) conditional used to extend the vocabulary by expressing new relations in terms of basic facts
Ex. $\text{Mother}(y, x) \Leftarrow \text{ChildOf}(x, y) \wedge \text{Woman}(y)$ [which is true $\forall x, y \in \text{Domain}$]

Parts of a Rule

- $Q \Leftarrow P$

Ex. $\text{Mother}(y,x) \Leftarrow \text{ChildOf}(x,y) \wedge \text{Woman}(y)$

- A rule has two parts:

- *Antecedent*: rule part that must be satisfied in order to the rule to be applicable

Ex. $\text{ChildOf}(x, y) \wedge \text{Woman}(y)$

- *Consequent*: (single) fact that is satisfied once the rule is applied

Ex. $\text{Mother}(y, x)$

- Ex. If our domain “knows” –antecedent– that Jane is a woman [$\text{Woman}(\text{jane})$] who has Jim as her child [$\text{ChildOf}(\text{jim}, \text{jane})$], then the rule is **applicable**. And, if we apply the rule, then the fact –consequent– that Jane is the mother of Jim [$\text{Mother}(\text{jane}, \text{jim})$] is incorporated to our domain knowledge (in the database of facts).

Sorts of Rules (according to the antecedent)

- **Conjunctive**: the antecedent is a conjunction of facts. All the single facts in the antecedent must be satisfied in order to the rule to be applicable.
 Ex. $\text{Can_vote}(x) \Leftarrow \text{Person}(x) \wedge \text{Adult}(x).$

- **Disjunctive forms**: the antecedent contains some disjunction
 - Disjunctive: $[b \Leftarrow a_1 \vee a_2 \vee \dots \vee a_k]$
 Ex. $\text{Can_vote}(x) \Leftarrow \text{Man}(x) \vee \text{Woman}(x)$
 - k-term DNF: $[b \Leftarrow (a_{11} \wedge \dots \wedge a_{1k1}) \vee \dots \vee (a_{i1} \wedge \dots \wedge a_{iki})], i \leq k.$
 Ex. $\text{Access_URV}(x) \Leftarrow (\text{Local}(x) \wedge \text{Younger}(x, 25) \wedge \text{CV_above}(x, 3)) \vee (\text{Local}(x) \wedge \neg \text{Younger}(x, 25)) \vee (\neg \text{Local}(x) \wedge \text{CV_above}(x, 3.5))$
 - k-DNF: $[b \Leftarrow (a_{11} \wedge \dots \wedge a_{1k1}) \vee \dots \vee (a_{i1} \wedge \dots \wedge a_{iki})], k_j \leq k.$
 Ex. Same as k-term-DNF where k restricts the no. of conjunctants in the clauses.
 - k-CNF: $[b \Leftarrow (a_{11} \vee \dots \vee a_{1k1}) \wedge \dots \wedge (a_{i1} \vee \dots \vee a_{iki})], k_j \leq k.$
 Ex. $\text{Accept_to_club}(x) \Leftarrow (\text{Profession}(x, \text{engineer}) \vee \text{Profession}(x, \text{doctor}) \vee \text{Profession}(x, \text{politician})) \wedge (\text{Income_above}(x, \text{€100K}) \vee \text{Properties_above}(x, \text{€3M}))$

Some Rule-Based representation problems

- There are some knowledge representation problems that may affect the reasoning process:
 - Problem 1: Alternative ways of representing the same knowledge
 - Problem 2: Repetition of searches
 - Problem 3: Order of goals
 - Problem 4: Order of rules

Problem 1: Alternative ways of representing the same knowledge

$\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,y).$

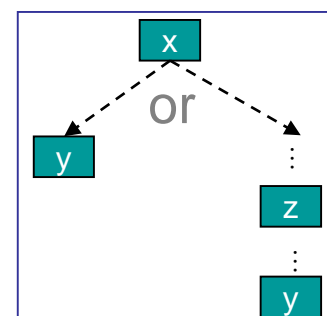
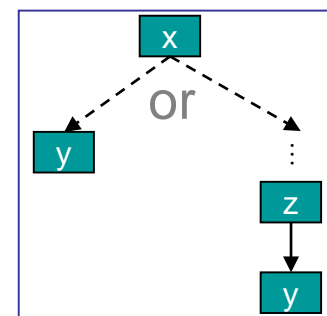
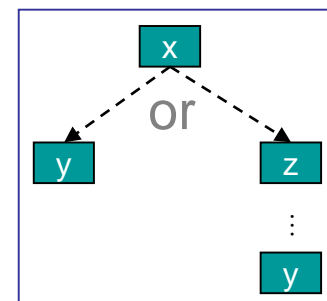
$\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,z) \wedge \text{Ancestor}(z,y).$

$\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,y).$

$\text{Ancestor}(x,y) \Leftarrow \text{Parent}(z,y) \wedge \text{Ancestor}(x,z).$

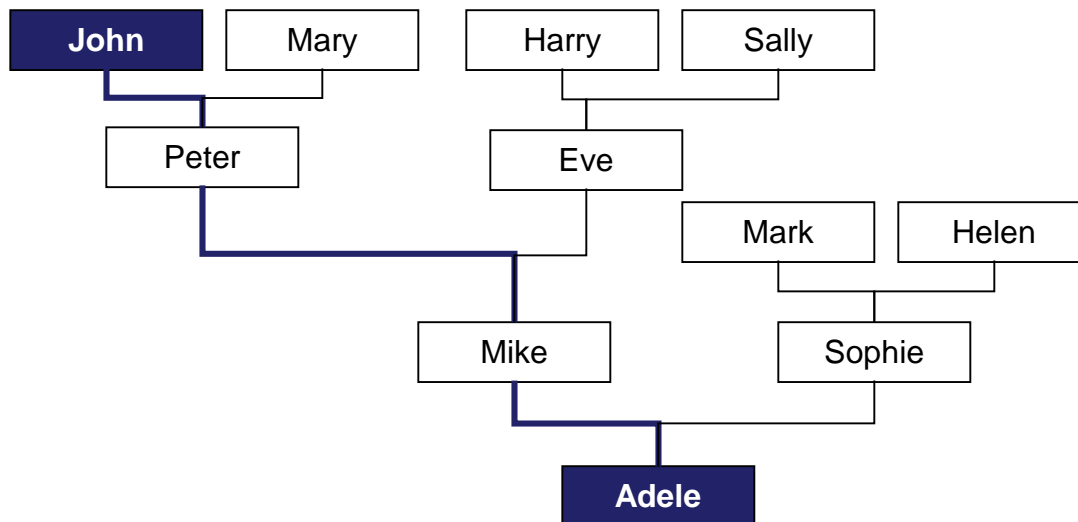
$\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,y).$

$\text{Ancestor}(x,y) \Leftarrow \text{Ancestor}(x,z) \wedge \text{Ancestor}(z,y).$



Reasoning time cost depends on knowledge representation 1

$\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,y).$
 $\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,z) \wedge \text{Ancestor}(z,y).$



Set of rules 1:

Query: $\text{Ancestor}(\text{john}, \text{adele})?$

$P(j,a)?$ **no**

or

$P(j,z) \ \& \ A(z,a)?$

$z = \text{Peter}!$ (only option)

$P(j,p)?$ **yes**

and

$A(p,a)?$

$P(p,a)?$ **no**

or

$P(p,z) \ \& \ A(z,a)?$

$z = \text{Mike}!$ (only option)

and

$P(p,m)?$ **yes**

$A(m,a)?$

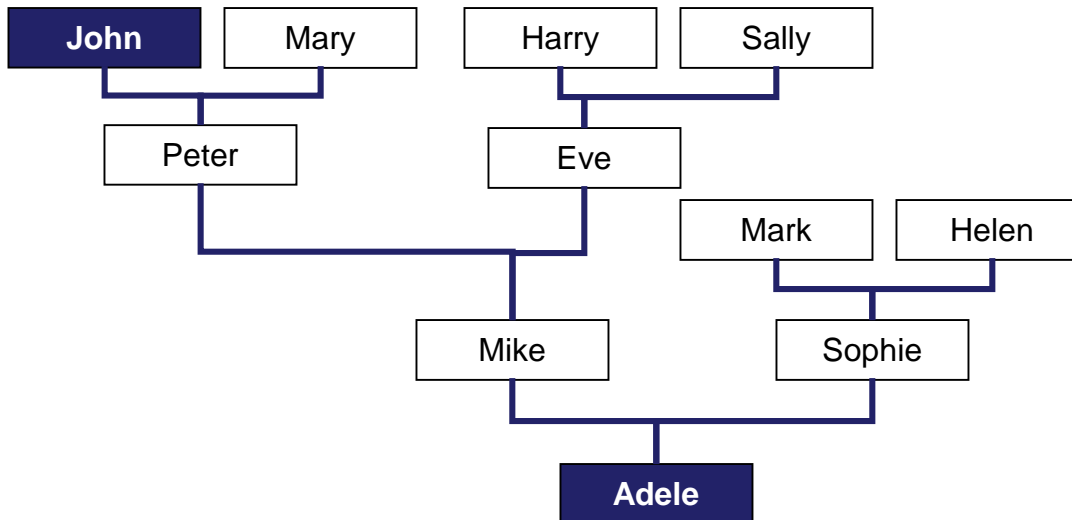
$P(m,a)?$ **yes**

Answer: *yes*

How many queries to the KB? 10

Reasoning time cost depends on knowledge representation 2

$\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,y).$
 $\text{Ancestor}(x,y) \Leftarrow \text{Parent}(z,y) \wedge \text{Ancestor}(x,z).$



Set of rules 2:

Query: *Ancestor(john,adele)?*

$P(j,a)?$ **no**

$P(z,a) \& A(j,z)?$

$z = \text{Mike or Sophie ! (two only options)}$

Suppose $z = \text{Mike !}$

$P(m,a)?$ **yes**

$A(j,m)?$

$P(j,m)?$ **no**

$P(z,m) \& A(j,z)?$

$z = \text{Peter or Eve ! (two only options)}$

Suppose $z = \text{Peter !}$

$P(p,m)?$ **yes**

$A(j,p)?$

$P(j,p)?$ **yes**

Suppose $z = \text{Eve !}$

$P(e,m)?$ **yes**

$A(j,e)?$

$P(j,e)?$ **no**

$P(z,e) \& A(j,z)?$

$z = \text{Harry or Sally ! (two only options)}$

Suppose $z = \text{Harry !}$

$P(h,e)?$ **yes**

$A(j,h)?$

$P(j,h)?$ **no**

$P(z,h) \& A(j,z)?$

$z = \text{NULL ! (there are not parents of Harry)}$ **no**

Suppose $z = \text{Sally !}$

$P(s,e)?$ **yes**

$A(j,s)?$

$P(j,s)?$ **no**

$P(z,s) \& A(j,z)?$

$z = \text{NULL ! (there are not parents of Sally)}$ **no**

Suppose $z = \text{Sophie !}$

$P(s,a)?$ **yes**

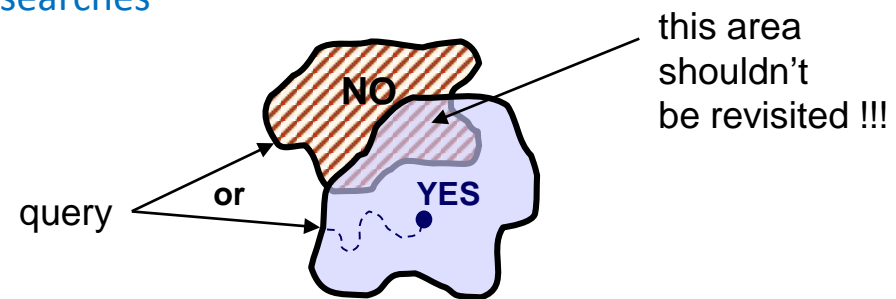
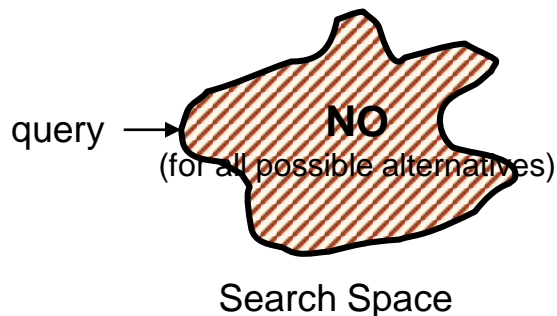
$A(j,s)?$

How many queries to the KB? It depends on the order of suppositions, but in the range [10, 35]

NOTICE: predicates with most restrictive terms first

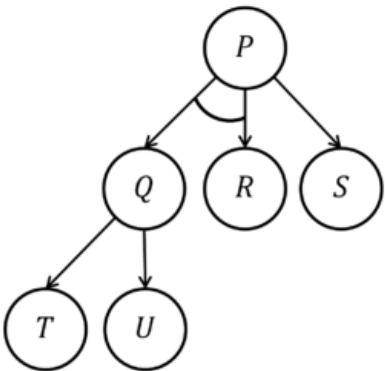
Problem 2: Repetition of seaches

- There are three **results** in a reasoning:
 - the result is **explicit** in the database
Ex. $\{..., \text{Rich}(\text{john}), ...\} \models \text{Rich}(\text{john})$
 - the result is **implicit** in the knowledgebase
Ex. $\{..., \text{Rich}(x) \leftarrow \text{Politician}(x) \wedge \text{Corrupt}(x), \text{Politician}(\text{john}), \text{Corrupt}(\text{john}), ... \} \models \text{Rich}(\text{john})$
 - the result is **NO** (**closed-world assumption**)
- Search Structures:
 - Search space**: region of all possible options, either valid or not.
 - And-or tree**: graphical representation of the reduction of problems (or goals) to conjunctions and disjunctions of subproblems (or subgoals).
- If the answer is NO, we must explore all the feasible alternatives in the search space, **avoiding repeated searches**
- If the answer is YES, we must **avoid repeated searches**



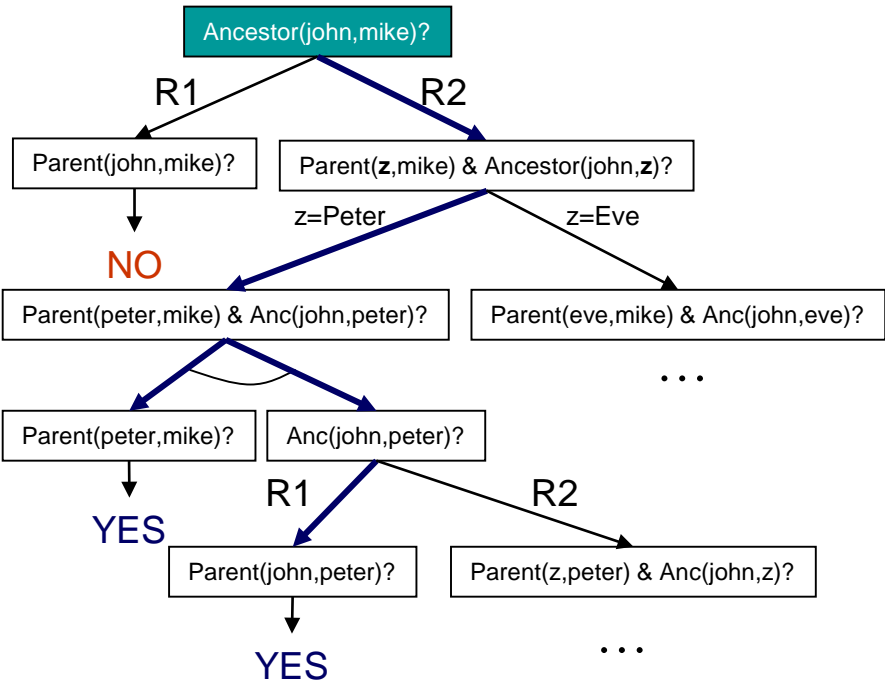
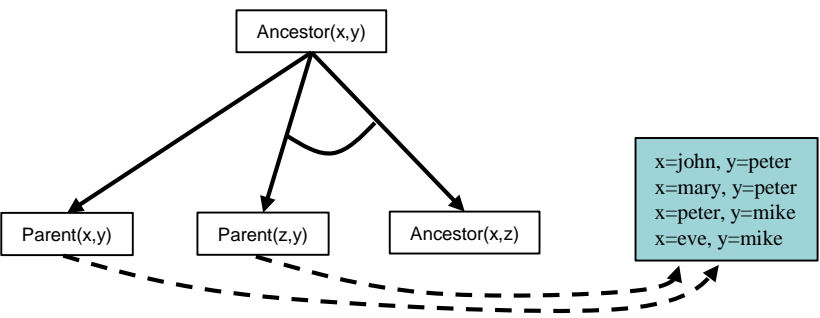
And-Or trees

$P \Leftarrow Q \wedge R$
 $P \Leftarrow S$
 $Q \Leftarrow T$
 $Q \Leftarrow U$



Parent(john,peter).
Parent(mary,peter).
Parent(peter,mike).
Parent(eve,mike).

R1: $\text{Ancestor}(x,y) \Leftarrow \text{Parent}(x,y).$
R2: $\text{Ancestor}(x,y) \Leftarrow \text{Parent}(z,y) \wedge \text{Ancestor}(x,z).$



Example: Fibonacci numbers 1,1,2,3,5,8,13,21,33,...



$$f(n) = f(n-1) + f(n-2)$$

F(0,1).

F(1,1).

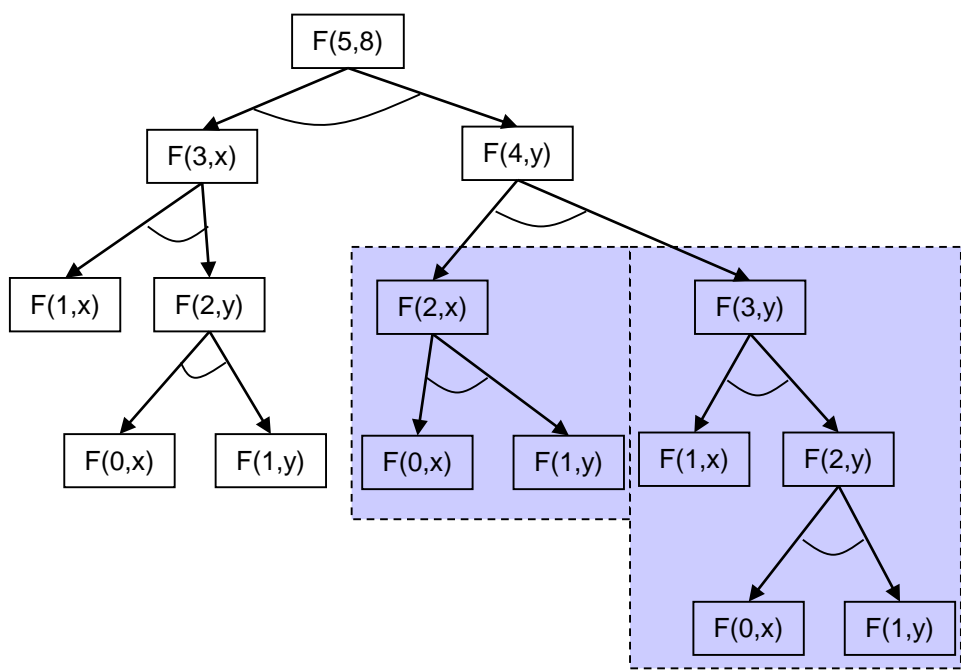
$F(+2(n),v) \Leftarrow F(n,x) \wedge F(+1(n),y) \wedge \text{Plus}(x,y,v).$

$F(n,v) \Leftarrow F'(n,1,0,v).$

$F'(0,y,z,y).$

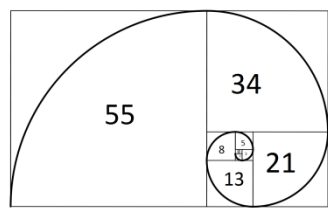
$F'(+1(n),y,z,v) \Leftarrow \text{Plus}(y,z,s) \wedge F'(n,s,y,v).$

Meaning: $F'(n,y,z,v)$
n = nth position
y = current Fibonacci number
z = previous Fibonacci number
v = nth Fibonacci number



15 searches

F(5,8)
F'(5,1,0,8)
F'(4,1,1,8)
F'(3,2,1,8)
F'(2,3,2,8)
F'(1,5,3,8)
F'(0,8,5,8)
YES



7 searches !

Problem 3: Order of goals

- $\text{AmericanCousin}(x,y) \Leftarrow \text{American}(x) \wedge \text{Cousin}(x,y)$
- Execution $\text{AmericanCousin}(x,\text{Sally})?$
- Equivalent to:
 - for all $x \in \text{Americans}()$
 - return $\text{Sally} \in \text{Cousins}(x)$
- One loop for each american: millions
- very SLOW
- $\text{AmericanCousin}(x,y) \Leftarrow \text{Cousin}(x,y) \wedge \text{American}(x)$
- Execution $\text{AmericanCousin}(x,\text{Sally})?$
- Equivalent to:
 - for all $x \in \text{Cousins}(\text{Sally})$
 - return $x \in \text{Americans}()$
- One loop for each cousin of Sally: tens
- very FAST

NOTICE: most restrictive clauses first

Problem 4: Order of rules

- $\text{UnivStudent}(x) \Leftarrow \text{BScStudent}(x)$
- $\text{UnivStudent}(x) \Leftarrow \text{MScStudent}(x)$
- $\text{UnivStudent}(x) \Leftarrow \text{PhDStudent}(x)$
- Execution: $\text{UnivStudent}(\text{Sally})$?
 - Sally a BSc Student: $|\text{BSc}|$: thousands
 - Sally a MSc Student: $|\text{BSc}| + |\text{MSc}|$: thousands
 - Sally a PhD Student: $|\text{BSc}| + |\text{MSc}| + |\text{PhD}|$: thousands
- very SLOW in all cases
- $\text{UnivStudent}(x) \Leftarrow \text{PhDStudent}(x)$
- $\text{UnivStudent}(x) \Leftarrow \text{MScStudent}(x)$
- $\text{UnivStudent}(x) \Leftarrow \text{BScStudent}(x)$
- Execution: $\text{UnivStudent}(\text{Sally})$?
 - Sally a BSc Student: $|\text{PhD}| + |\text{MSc}| + |\text{BSc}|$: thousands
 - Sally a MSc Student: $|\text{PhD}| + |\text{MSc}|$: hundreds
 - Sally a PhD Student: $|\text{PhD}|$: tens
- FAST in some cases

NOTICE: most restrictive rules first (or more *priority*)

Sorts of Rule-Based Reasoning

- **IF P THEN Q**

Ex. IF X is an adult THEN X can vote [$\text{Can_vote}(x) \Leftarrow \text{Adult}(x)$]

- Forward Chaining: sort of reasoning that goes from what we know (P) to what we want to know (Q).

Ex. if we know that John is an adult, we can conclude that he can vote

- Backward Chaining: sort of reasoning that goes from what we want to know (Q) to the premises that we know (P)

Ex. if we want to know if John can vote, we must check whether he is an adult or not

- Data-Directed Reasoning: sort of reasoning that goes from assertions of P to assertions of Q

Ex. if our DB contains that John is an adult (it was asserted so), we extend the DB to contain that he can vote.

Notice that, here, the DB contains the data (facts=data) that we are finding out.

- Goal-Directed Reasoning: sort of reasoning that goes from goals of Q to goals of P

Ex. if our **set of goals** contains that John can vote, then we can replace this goal by the goal “is he an adult?”

Notice that, here, the DB contains goals (facts=goals) that have to be satisfied.

Production Systems

- **Definition** (*Production System*): a PS is a forward chaining system that uses rules of a certain form called **production rules (PR)** as its representation of general knowledge. It has a dynamic memory called **working memory (WM)** that evolves as rules are applied.
- **Definition** (*Production Rule*) a PR is a two-part structure comprising an **antecedent** and a **consequent**. If the set of (and) conditions in the antecedent are true, then the set of actions in the consequent are applied.

IF conditions THEN actions

- The conditions in the antecedent are evaluated with the current working memory, following a three-step cycle:
 1. **recognize active rules**: find out the set of rules that can be applied.
 2. **resolve conflicts**: select the rule to be applied among the active rules.
 3. **trigger rule**: change the WM according to the actions of the triggered rule.
 4. loop steps 1-4

Definitions

- **Definition** (*Working Memory*): the WM is a set of working memory elements (WME).
- **Definition** (*Working Memory Element*): is a n-tuple of the form

$$(type\ att_1: val_1 \dots att_n: val_n)$$

where $type$, att_i , val_i are atoms.

Ex. properties like (person age: 27 name: John home: Tarragona)

Ex. relationships like (fact relation: married who: John to: Mary)

- The order of the attributes in a WME is irrelevant.

Production Rule Antecedents

- The antecedent of a production rule is a set of positive or negative conditions of the form $(type\ att_1: spec_1 \dots att_m: spec_m)$ where $type$, att_i are atoms, and $spec_i$ is either:
 1. an atom
 2. a variable
 3. an evaluable expression within square brackets []
 4. a test within curly brackets { }
 5. a conjunction/disjunction/negation of a specification
- Ex. (person name: John occupation: x age: [n+4])
- Ex. $\neg(\text{person age: } \{<23\} \wedge \{>6\})$

Production Rule Consequents

- The consequent of a production rule is a **sequence of actions** to be executed from left-to-right. Each action can be of one of the following forms:
 1. **ADD *pattern***: a new WME specified in pattern is added to the WM.
 2. **REMOVE *i***: remove from the WM the WME that matches the *i*-th condition in the antecedent of the rule.
 3. **MODIFY *i (att spec)***: modify the WME in the WM that matches the *i*-th condition in the antecedent of the rule by replacing the current value of *att* by *spec*.
- Notice: REMOVE and MODIFY are not applicable if the condition is negative.
- Ex. IF (student name: x) THEN ADD (person name: x)
- Ex. IF (person age: x name: n) (birthday who: n date: d) (today date: d) THEN
 MODIFY 1 (age [x+1])
 REMOVE 2

Production System Example

- **The world of blocks: stack bricks in size order**

- Knowledge Base:

IF (brick position: heap name: n size: s)

- (brick position: heap size: {>s})

- (brick position: robot-hand)

THEN MODIFY 1 (position robot-hand)

IF (brick position: robot-hand)

(counter value: i)

THEN MODIFY 1 (position i)

MODIFY 2 (value [i+1])

- Working Memory

(counter value:1)

(brick name: A size: 10 position: heap)

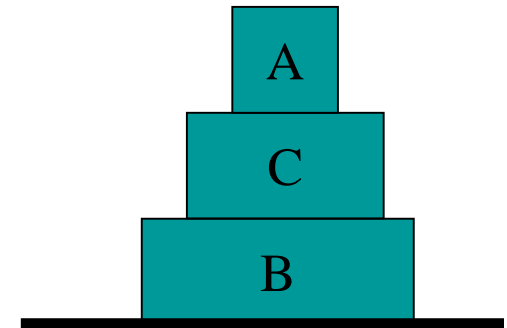
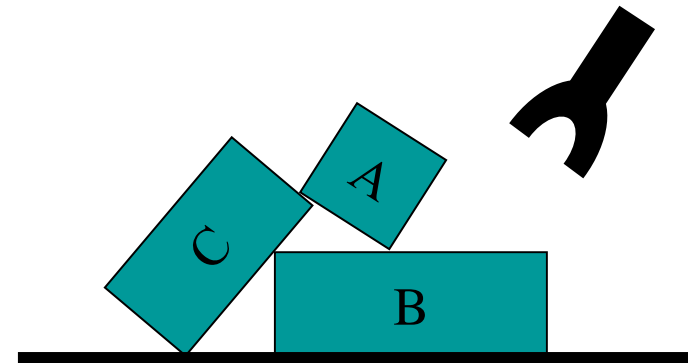
(brick name: B size: 30 position: heap)

(brick name: C size: 20 position: heap)

- After triggering the first rule

...

(brick name: B size: 30 position: robot-hand)



Conflict Resolution Strategies

- If there are several rules that can be triggered at the same time, we may provide a means to select one.
- There are several strategies:
 1. *Random*: take one at random.
 2. *Order*: take the first one in the knowledge base.
 3. *Specificity*: take the most specific rule.
 - R1 more specific than R2 iff $WME(R2) \supset WME(R1)$
 - R1 more specific than R2 iff $\#WME(R2) > \#WME(R1)$
 4. *Recently*: depending on the last time the rule was triggered
 - *most-recent-rule*: take the rule that has been more recently triggered
 - *most-recent-WME*: take the rule that matches the most recently created or modified WME.
 - *least-recent-rule*: opposite of most-recent-rule
 - *least-recent-WME*: opposite of most-recent-WME
 5. *Refractoriness*: avoid using the rule that has just been applied with the same values in its variables.

CLIPS

- A tool for building expert systems
- <http://www.clipsrules.net/>



CLIPS Production Rules

- Facts: (fact-name att1 att2 ... attN)
- Actions: (**assert** fact ...), (**retract** fact-id ...)
- List of facts: (**facts**)
- (**defrule** name [comment] [priority] premise => conclusion)
- List of rules: (**rules**)
- Comment: "String" describing the rule
- Priority: (**declare** (**salience** num)), bigger num => more priority
- Premise:
 - Exists: (fact-name restriction1 restriction2 ... restrictionN)
 - Does not exist: (**not** (fact-name restriction1 restriction2 ... restrictionN))
- Restriction:
 - No-named: ?
 - Named: ?name
 - Conditioned: ?name&:condition
- Condition Examples: (< ?s 50), (>= ?s ?s2), ...
- Conclusion: list of actions
- Execution: (**run**) or (**run** number)
- Showing messages: (**printout** t thing ...)

CLIPS Production Rules: The World of Blocks

- Facts

(counter NUMBER)

(brick NAME SIZE POSITION)

- R1: Pick up the greatest brick

(defrule R1

 ?p1 <- (brick ?n ?s heap)

 (not (brick ? ?size&:(> ?size ?s) heap))

 (not (brick ? ? ?place&:(eq ?place robot-hand)))

 =>

 (retract ?p1)

 (assert (brick ?n ?s robot-hand)))

- R2: Stack the robot-hand picked brick

(defrule R2

 ?p1 <- (brick ?n ?s robot-hand)

 ?p2 <- (counter ?i)

 =>

 (retract ?p1 ?p2)

 (assert (brick ?n ?s ?i) (counter (+ 1 ?i))))

- Execution:

(deffacts initial (counter 1) (brick A 10 heap) (brick B 30 heap) (brick C 20 heap))

(reset)

(run 1)

(facts)

CLISP Templates

- (deftemplate name [comment] slot ...)
- Slot:
 - Single value: (slot slot-name)
 - Multiple values: (multislot slot-name)

CLIPS Templates: The World of Blocks

- Template:

```
(deftemplate brick (slot name) (slot size) (slot position))
```

- R1: Pick up the greatest brick

```
(defrule R1
  ?p1 <- (brick (size ?s) (position heap))
  (not (brick (size ?s2 & (> ?s2 ?s)) (position heap)))
  (not (brick (position ?place & (eq ?place robot-hand))))
  =>
  (modify ?p1 (position robot-hand)))
```

- R2: Stack the robot-hand picked brick

```
(defrule R2
  ?p1 <- (brick (position robot-hand))
  ?p2 <- (counter ?i)
  =>
  (modify ?p1 (position ?i))
  (retract ?p2)
  (assert (counter (+ 1 ?i))))
```

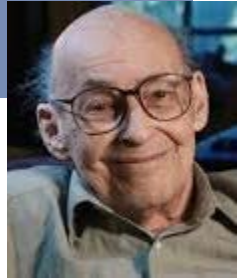
- Execution:

```
(defacts initial (counter 1) (brick (name A) (size 10) (position heap)) (brick (name B) (size 30) (position heap)) (brick (name C) (size 20) (position heap)))
(reset)
(run 1)
(facts)
```

2.3. Object-Oriented Representation

- In previous KR models:
 - The units of knowledge are self contained (GOOD).
 - The knowledge about domain objects (or object types) is scattered all around the knowledge base (BAD).
 - Reasoning cannot be controlled
- These limitations can be overcome with a knowledge representation oriented to **objects**.
- In 1974, Marvin Minsky proposes the **frame** model.
- Since then, multiple frame systems have appeared:
 - at *Stanford University* (1970's): UNIT, STROBE, CLASS, RLL, CYCL, ARLO, THEO, JOSIE, OPUS, KEE, KAPPA, ...
 - at *Harvard University* (1970's): KL-ONE, NIKL, KANDOR, KL-TWO, K-REP, KREME, BACK, SB-ONE, KRYPTON, LOOM, CLASSIC, ...
 - at *Carnegie Mellow University* (1980's): SRL, FRAMEKIT, PARMENIDES, KNOWLEDGECRAFT, ...

Frames (Minsky, 1974)



- Knowledge structure used to represent objects and object types.
- There are two types of frames:
 - individual (or instance) frames: they represent single objects (ex. John)
 - generic (or class) frames: they represent categories of objects (ex. person)
- Individual and generic frames share the same structure:

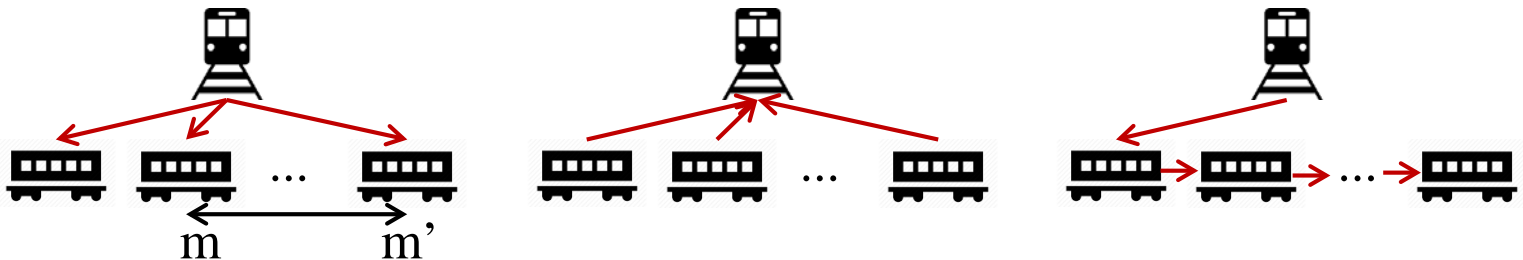
```
(FrameName  
  <property1>  
  ...  
  <propertyn>  
)
```

```
(Tarragona  
  <:population 200000>  
  <:climate "mediterranean">  
  <:has-coast? YES>  
)
```

```
(City  
  <:population Number>  
  <:climate ClimateType>  
  <:has-coast? Boolean>  
)
```

Slots

- Frame properties are called **slots**.
- Slots can be assigned with **fillers** (these are the values contained in the slots).
- Types of Slots:
 1. **Generalization (2)**: The slot **:INSTANCE-OF** determines the class of an individual frame (\in) (e.g., Tarragona :INSTANCE-OF City). The slot **:IS-A** determines the super-class of a generic frame (\subseteq) (e.g., City :IS-A SocialOrganization). The main frame is called a **specialization** of the frame in the slot :IS-A.
 2. **Agregation (1)**: The slot **:PART-OF** is used to state the parts of a frame (e.g., Building :PART-OF City, or Tail :PART-OF Dog).
 3. **Association (n)**: Other slots define associations with a meaning in both direction (e.g., John :LIVES-IN Tarragona).
- Storing multiple values:
 1. **Multivalued slot**: e.g., (Train <:wagons [MIN-CARDINALITY m] [MAX-CARDINALITY m'] Wagon>)
 2. **Inverse slot**: e.g., (Wagon <:PART-OF Train>) –each wagon points to one train
 3. **Chaining values**: e.g., (Train <:wagons Wagon>) (Wagon <:next-wagon Wagon>)



Facets

- A slot may have properties called **facets**.
- Facets contain information about the slot.
- Special Facets:
 - **Facet IF-ADDED** attaches a procedure to be run when the value of the slot is set (Ex. <Parent [IF-ADDED {check-this-person-has-siblings}]]>)
 - **Facet IF-NEEDED** attaches a procedure to be run when the value of the slot is asked or used (Ex. <NDescendants [IF-NEEDED {calculate-Num-Descendants}]]>)
 - Other possible facets: **DEFAULT**, **DOMAIN**, **MAX/MIN-CARDINALITY**, ...

Reasoning with Frames

- Much of the reasoning with a frame system involves
 - **creating** individual instances of generic frames,
 - **filling** some of the slots with values, and
 - **inferring** some other values.
- In this process,
 - **inheritance** is a concept that plays an important role.
 - introducing the filler of a slot follows the **assignment** loop
 - requiring the filler of a slot follows the **requirement** loop
- Reasoning follows the **reasoning** loop.

Conclusion: frame reasoning = inheritance + 3 loops

Inheritance

- **Inheritance of properties** is the process of passing information from generic frames down through their specializations and eventually to their instances.
- Frame systems implement **defeasible inheritance** (i.e., we use inherited values only if we cannot find a filler otherwise).
- **Multiple inheritance**: in some frame systems frames are allowed to be instances or specializations of more than one generic frame and they inherit slots from all of them (ex. John is an instance of *person*, but also of *student*).
- Inherited slots may be assigned a different filler (ex. the color of elephants can be grey, but there is a refinement of elephants that are white).

```
(Population
  <:inhabitants Number>
  <:major Person>
  <:has-coast? Boolean>
  <:climate ClimateType>
  <:perCapitaIncome [IF-NEEDED calc INCOME/inhab]>
)
```

```
(City
  <:is-a Population>
  <:inhabitants [IF-ADDED chech>10K]>
  <:climate ClimateType>
  <:has-coast? Boolean>
  <:has-train? yes>
)
```

```
(MediterraneanCity
  <:is-a City>
  <:is-a TuristicDestination>
  <:climate Mediterranean>
  <:has-coast? no>
)
```

```
(Tarragona
  <:instance-of MediterraneanCity>
  <:major CurrentMajor>
  <:has-coast? yes>
  <:NoTuristsPerYear 2000000>
)
```

Assignment Loop (WRITE)

When a value is assigned to an instance frame slot:

1. the value becomes the filler of the slot.
2. if there is an IF-ADDED facet, its procedure is triggered.
3. Otherwise, the hierarchy of frames is followed up to find the first IF-ADDED procedure for the slot. If found, trigger this procedure, otherwise do nothing.

Triggering an IF-ADDED procedure may cause other slots to be filled or new frames to be instantiated and the loop is repeated.

1. Tarragona reaches 15K inhabitants (`Tarragona:inhabitants := 15K`)
2. This value is stored in the frame Tarragona slot inhabitants
3. The if-added daemon that checks that the No inhabitants of cities are above 10000 is found
4. The system checks correctness that $15K > 10K$
(if it was not, the daemon could assign a different value, e.g. "unknown")
- (`City ... :inhabitants [IF-ADDED (lambda(v)(if (< v 10000) "unknown" v))]` ...)

Requiring Loop (READ)

When a value is required from an instance or generic frame slot:

1. if there is a filler in the slot, then that value is returned.
2. otherwise, if there is an IF-NEEDED procedure, it is triggered. If there is none, the hierarchy of frames is followed up to find the first IF-NEEDED procedure for the slot. If found, trigger this procedure, otherwise do nothing. The value returned by the corresponding IF-NEEDED function is returned.
3. If neither of these produce a result, the value is considered “unknown”.

Triggering an IF-NEEDED procedure may cause other slots to be filled/read or new frames to be instantiated and the loop is repeated.

1. Is Tarragona by the coast? “yes” after the slot has-coast? filler.
2. What’s the climate of Tarragona? Mediterranean after inheritance from MediterraneanCity frame.
3. What is the percapita income in Tarragona? the value provided by the daemon IF-NEEDED inherited from the frame Population.

The Reasoning Loop

1. The user of a frame system declares that an object exists by instantiating some generic frame.
2. All slot fillers of the new instance frame that are not provided explicitly by the user, but can be inherited are inherited.
3. For all the slots with a filler, the IF-ADDED procedure is run. This may cause other slots to be filled, or new frames instantiated, and the cycle repeats.

1. Girona frame is created as an instance-of MediterraneanCity
2. If nothing is said about the slot :has-coast?, the filler "no" of MediterraneanCity is assigned.
3. The has-train? slot is also assigned the value "yes".
4. If the No of inhabitants of Girona is set to 150000, then the IF-ADDED daemon is triggered.
5. This daemon could be used to fill in the PerCapitaincome slot, which could trigger on its turn another IF-ADDED daemon.

Lisp-like notation for *daemons*



- (lambda (v) *body*) in :IF-ADDED and :IF-NEEDED facets:
 - :IF-ADDED: after the value v is stored in the slot, the lambda function is runned.
 - :IF-NEED: before the value v stored in the slot is given, the lambda function is runned and the result given.
- The *body* computes a resultant value:
 - Preorder notation: (+ 5 4), (< 10 15), ...
 - Assignment: (**set** slot value)
 - Conditional: (**if** cond then [else])
 - Loop: (**while** cond body)
 - Environments: (**let** ((var value)(var value)...) body-accessing-var's)
 - Grouping sentences: (sent1 ... sentN) This group takes value the value of last sentence.
- **SELF**:slot-name is used to access the value of slot-name in this frame.
- Frame-name:slot-name is used to access the value of slot-name in Frame-name.
- Examples:
 - Percentage: (lambda (v) (* v 100))
 - Checking: (lambda (v) (if (is-even v) v "unknown"))
 - Correcting: (lambda (v) (if (= v:IS-A Person) v Anonymous-person))
 - Calculating: For slot-name
 - (lambda (v) (if (= SELF:slot-name "unkown") v (+ SELF:slot-name v)))

Example 1: Classic Cars

<http://www.topclassiccars.com/>


(ClassicCar

```

<:IS-A      Car>
<:Company   CarCompany>
<:Model     [IF-ADDED {
              (lambda (x)
                (if (not (producesModel? SELF:Company x))
                    "Unknown" x))
              }]>
<:HorsePower [range 50..200]>
<:StartProd  Date>
<:FinishProd Date>
<:Color      [range {R W B Y DARK OTHER}]>
<:FactoryPrice Number>
<:RetailPrice [IF-NEEDED { (lambda (x)
                            (* SELF:FactoryPrice 1.3)//add 30% interests
                            )}]
              [IF-ADDED { (lambda (x)
                            (if (< (* SELF:FactoryPrice 1.15) x)
                                "Unkown" //price must be above 15% of FP
                            )}]>
<:NWheels    4>

```

)



(John'sBeetle

```

<: INSTANCE-OF
<: Company
<: HorsePower
<: StartProd
<: Color
<: FactoryPrice

```

)

```

ClassicCar>
Volkswagen>
63>
1938>
B>
10000€>

```

Example 2: Traveling

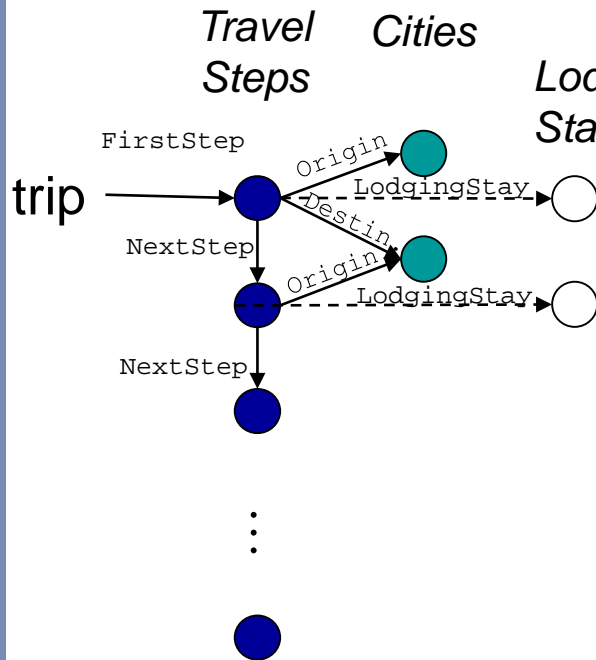
```
(Trip
  <:FirstStep TravelStep>
  <:Traveler Person>
  <:BeginDate Date>
  <:EndDate Date>
  <:Cost Price>
)
```

```
(TripPart
  <:BeginDate Date>
  <:EndDate Date>
  <:Cost Price>
  <:PaymentMethod FormOfPayment>
)
```

```
(LodgingStay
  <:IS-A TripPart>
  <:Place City>
  <:LodgingPlace LodgingPlace>
  <:ArrivingTravelStep Date>
  <:Departing Date>
```

Lodging
Stay (Tra

```
(TravelStep
  <:IS-A TripPart>
  <:Origin City>
  <:Destination City>
  <:LodgingStay LodgingsStay>
  <:Means TransportMean>
  <:DepartTime Date>
  <:ArrivalTime Date>
  <:NextStep TravelStep>
  <:PreviousStep TravelStep>
)
```



- I use to go by plane:

```
(MyTravelStep
  <:IS-A TravelStep>
  <:Means airplane>
)
```

- Origin of my trips Tarragona

```
<:Origin
[IF-NEEDED { (lambda (x)
  (if (null SELF:PreviousStep)
    Tarragona
    SELF:PreviousStep:Destination
  ))}]>
```

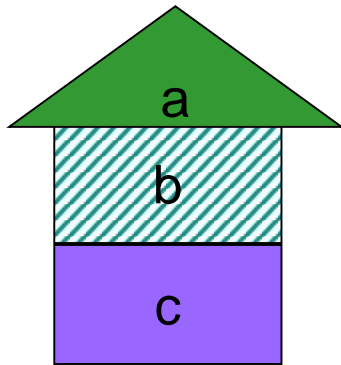
- Trip costs calculation

```
<:Cost
[IF-NEEDED { (lambda (x)
  (if (not (null x)) x
      (let ((c 0)
            (x SELF:FirstStep))
        (while (not (null x))
          (set c (+ c x:Cost))
          (set y x:LodgingStay)
          (if (not (null y))
              (set c (+ c y:Cost)))
              (set x x.NextStep))) c ))
}]>
```

Example 3: Blocks World

```
(Block
  <:Color BlockColor>
  <:On Block>
  <:Free Boolean>
)
```

```
(Pyramid
  <:IS-A block>
  <:Free false>
)
```



```
(A
  <:INSTANCE-OF Pyramid>
  <:Color green>
  <:On B>
)
```

```
(B
  <:INSTANCE-OF block>
  <:Free false>
  <:On C>
)
```

```
(C
  <:INSTANCE-OF block>
  <:Color magenta>
)
```

- When a block is stacked, the other block is not free

```
<:On [IF-ADDED {
  (lambda (x)
    (set x:Free false))}]>
```

- Blocks can only stack on free blocks

```
<:On [IF-ADDED {
  (lambda (v)
    (if (v:IS-A Block)
        (if (v:Free)
            ((set v:Free false)
             v))
        "unknown"))
}]>
```

Create and Remove frames dynamically

- IF-ADDED and IF-NEEDED facets can be used to:
 - create new frames
 - remove already existing frames.
- **SELF** is the way to refer to the current frame.
- There is an instruction to create frames that returns the frame:

```
(create frameName (  
    (:slot1 value1)  
    . . .  
    (:slotn valuen)  
))
```

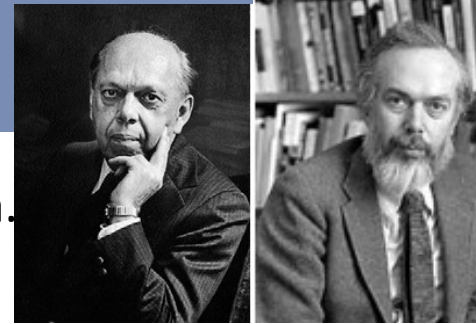
- There is an instruction to destruct frames that returns null:

```
(delete frame)
```

- Ex. “If I have a friend ill, I cancel my next visit and go visit him/her”

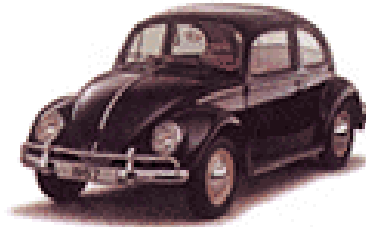
```
(if (= SELF:Friend:State ill)  
    (delete SELF:NextVisit)  
    (set SELF:NextVisit (create Visit (  
        (:from SELF)  
        (:to SELF:Friend))))))
```

Scripts (Schank & Abelson, 1977)



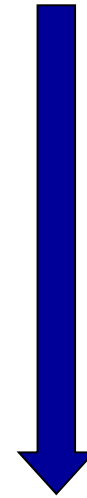
- Representing *procedural knowledge* with a frame system.
- In 1977, Schank & Abelson propose the *script* model.
- **Script**: A structure that describes appropriate sequences of events in a particular context. A type of frame that describes what happens temporally (know-how).
- **slot Prop(ertie)s**: objects being part of the script (frames or labels).
- **slot Roles**: agents involved in the script definition (frames or labels).
- **slot Starting/Opening conditions**: conditions that make the script to be valid (pre-condition).
- **Scene slots**: actions in the script.
- **slot Results**: conditions that are valid after the script is run (post-condition).
- Scripts extend frames with complex temporal/sequential events.

Example 1: Buy a Car



```
(BuyACar
  <:IS-A PurchaseProcedure>
  <:Props { :Shop :Car :Catalog :Office }>
  <:Roles { :Customer :Seller }>
  <:OpeningConditions {(SELF:Customer:WantsACar)}>
  <:Results {(set SELF:Car:Owner SELF:Customer)
             (set SELF:Customer:Money
              (- SELF:Customer:Money SELF:Car:RetailPrice))
             (set SELF:Customer:WantsACar false)
             (set SELF:Shop:Sells (+ 1 SELF:Shop:Sells)))}>
  <:Scene {Entering
           {(set SELF:Customer:Place SELF:Shop)
            (set SELF:Scene Inspecting)}
           Inspecting
           {(set SELF:Customer:Observes SELF:Car)
            (if (SELF:Customer:LikesCar)
                (set SELF:Scene Asking)
                (set SELF:Scene Leaving)))}
           Asking
           {(set SELF:Seller (get-free-seller SELF:Office))
            ...
           }
           Buying {...}}>
)
```

```
(John
  <:INSTANCE-OF Customer>
  ...
  <:WantsACar [IF-ADDED {
    (lambda (x)
      (if (x) --John wants a car
          (create BuyJohn'sCar (
            (:instance-of BuyACar)
            (:Shop Shop30)
            (:Car John'sBeetle)
            (:Catalog c889001)
            (:Office WV124)
            (:Customer SELF)
            (:Seller unknown)
          ))))}]>
  ...
)
```



```
(BuyJohn'sCar
  <:INSTANCE-OF BuyACar>
  <:Props {Shop30 John'sBeetle c889001 WV124}>
  <:Roles {John unknown}
)
```

Example 2: Restaurant



■ Sequence:

- Enter a restaurant
- Read the menu
- Order the meal
- Get the food
- Eat
- Ask the bill
- Pay
- Leave the restaurant

```
(Restaurant
  <:IS-A Script>
  <:Props    { :Street :Table :Menu-book :Course :Bill }>
  <:Roles    { :Customer :Waiter }>
  <:OpeningConditions { (SELF:Customer:isHungry)
                        (SELF:Restaurant:isOpen) }>
  <:Results  { (set SELF:Customer:isHungry false)
                (set SELF:Customer:Place Street)
                (set SELF:Customer:Money
                  (- SELF:Customer:Money SELF:Bill:Amount)) }>

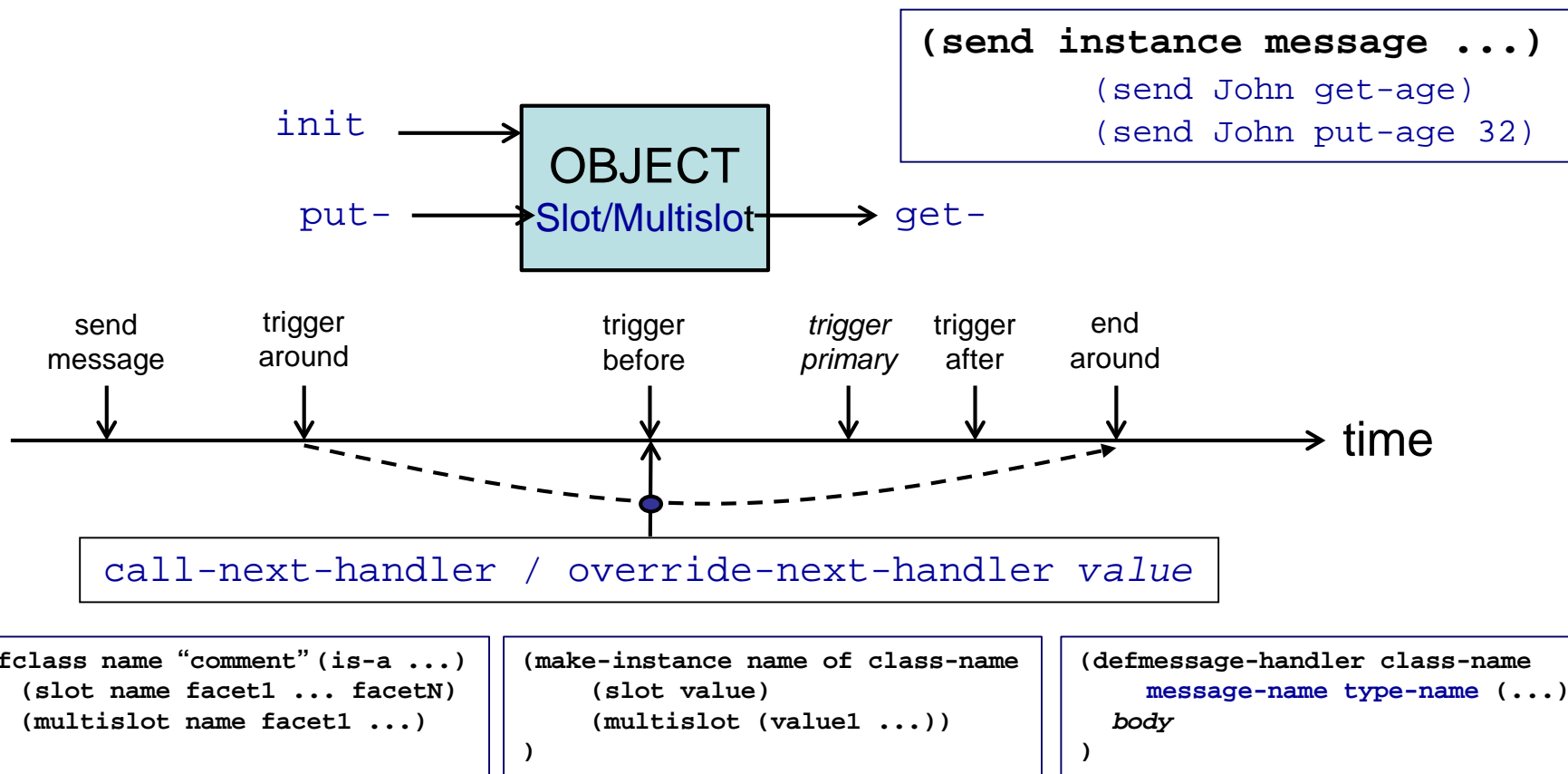
  <:Scene { Enter "Customer enters rest. and asks for menu"
            { (set SELF:Customer:Place SELF:Table)
              (set SELF:Menu-book:Used SELF:Customer)
              (set SELF:Scene Read-Menu) }
            Read-Menu "Customer reads the menu and orders"
            { (set SELF:Course
                  (select SELF:Customer SELF:Menu-book))
              (set SELF:Waiter:Order SELF:Course)
              (set SELF:Scene Get-Food) }
            Get-Food "Customer gets food and eats"
            { (set SELF:Course:Served SELF:Customer)
              (set SELF:Customer:Eats SELF:Course)
              (set SELF:Scene Ask-Bill) }
            Ask-Bill "Customer asks bill and pays"
            { (set SELF:Bill:asked-by SELF:Customer)
              (set SELF:Bill:state "payed" ) } }>
)
```


The Frame Representation Language at MIT

- “*The FRL Manual*”, R. Bruce Roberts, Ira P. Goldstein, 1977.
- Lisp representation of Frames.
- Elements: frames, slots, facets, datum, comment, messages.
- Frames:
 - `(frame-name ((slot-name (facet-name (datum)*))*))*`
 - `(FASSERT name slot1 ... slotN)` to create a frame.
- Slots:
 - AKO slot (A Kind Of): define frame-superframe relationships.
 - INSTANCE slot: define instance-generic frames relationships.
- Facets:
 - \$VALUE: contains the value of the slot.
 - \$DEFAULT: contains the default value of the slot.
 - \$IF-ADDED, \$IF-REMOVED, \$IF-NEEDED: define the demons related to a slot in case the filler of the slot is inserted, removed, or read, respectively.
- Accessing to slot facets:
 - `(FGET frame slot facet)`: returns the list of all the data items in facet or slot in frame.
 - `(FPUT frame slot)`, `(FPUT frame slot facet)`, `(FPUT frame slot facet datum)`: adds the last argument a the point of the frame named by the indicator path (previous arguments).

COOL

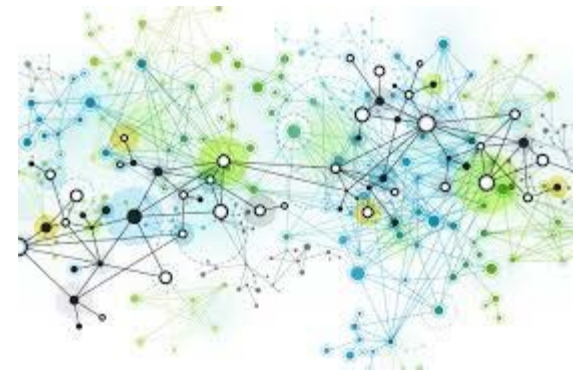
- CLIPS Object-Oriented Language
- <https://www.csie.ntu.edu.tw/~sylee/courses/clips/bpg/node9.html>



message-name: init, put-slot-name, get-slot-name, delete
type-name: before, around, after, primary

2.4. Network Representation

- In frames:
 - The units of knowledge are self contained (GOOD).
 - The knowledge about domain objects (or object types) is concentrated all around the objects (GOOD).
 - Knowledge is about establishing and exploiting **semantic relations** between objects, but frames are not designed to do that (BAD).
- This last problem can be solved with an orientation to represent knowledge as **semantic networks**.

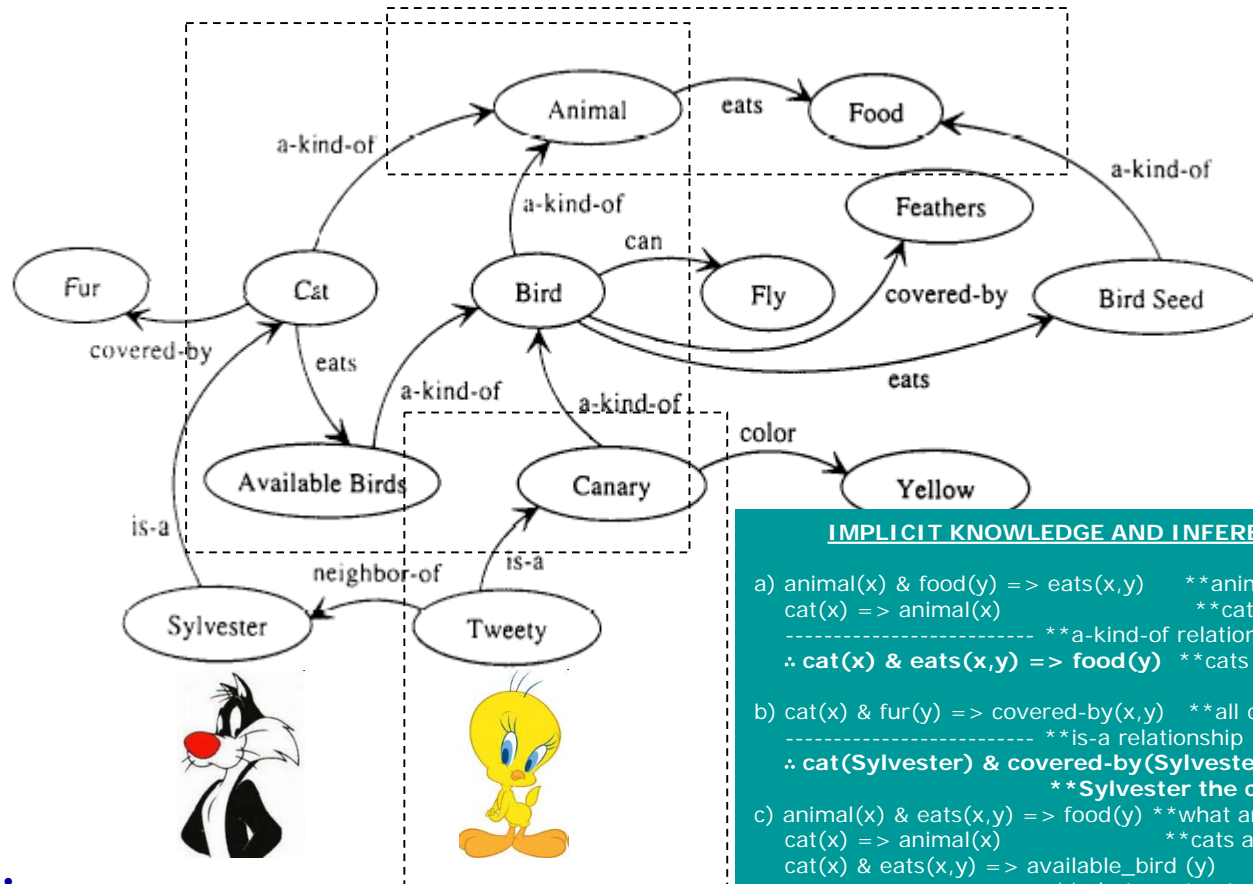


Sorts of Semantic Networks (John F. Sowa, 1992)

- **Definitional networks** emphasize the is-a relation between concepts. The resulting network, also called a generalization or subsumption hierarchy, supports the rule of inheritance for copying properties defined for a supertype to all of its subtypes. Since definitions are true by definition, the information in these networks is often assumed to be necessarily true.
- **Assertional networks** are designed to assert propositions. Unlike definitional networks, the information in an assertional network is assumed to be contingently true, unless it is explicitly marked with a modal operator. Some assertional networks have been proposed as models of the conceptual structures underlying natural language semantics.
- **Implicational networks** use implication as the primary relation for connecting nodes. They may be used to represent patterns of beliefs, causality, or inferences.
- **Executable networks** include some mechanism, such as marker passing or attached procedures, which can perform inferences, pass messages, or search for patterns and associations.
- **Learning networks** build or extend their representations by acquiring knowledge from examples. The new knowledge may change the old network by adding and deleting nodes and arcs or by modifying numerical values, called weights, associated with the nodes and arcs.
- **Hybrid networks** combine two or more of the previous techniques, either in a single network or in separate, but closely interacting networks.

Limitations:

1. representing negations
2. representing n-ary relationships



```
a) animal(x) & food(y) => eats(x,y)    **animals eat food
cat(x) => animal(x)                    **cats are animals
----- **a-kind-of relationship (inheritance1)
∴ cat(x) & eats(x,y) => food(y)      **cats only eat food

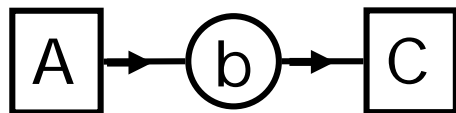
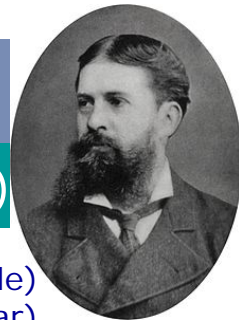
b) cat(x) & fur(y) => covered-by(x,y)  **all cats are fur covered
----- **is-a relationship (inheritance 2)
∴ cat(Sylvester) & covered-by(Sylvester,G) => fur(G)
      **Sylvester the cat is fur covered

c) animal(x) & eats(x,y) => food(y)    **what an animal eats is food
cat(x) => animal(x)                    **cats are animals
cat(x) & eats(x,y) => available_bird(y)
----- **what a cat eats is available_birds
∴ available_bird(x) => food(x)         **available_birds are food

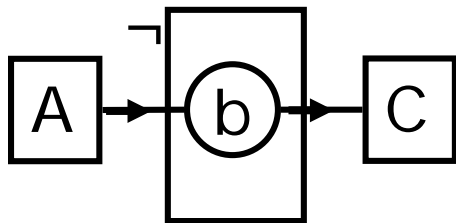
d) cat(Sylvester)    **Sylvester is a cat
-----
∴ available_bird(Tweety) => food(Tweety)
    **if Tweety is-a available_bird, then it becomes food (for Sylvester)
```

Knowledge Representation: “Assertional” Semantic Networks

(Charles S. Peirce, 1909)



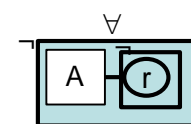
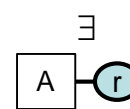
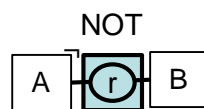
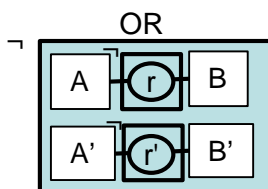
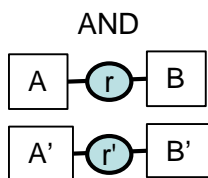
$\exists A \exists C: b(A, C)$ Ex. owns (John, Beetle)
owns (Person, Car)



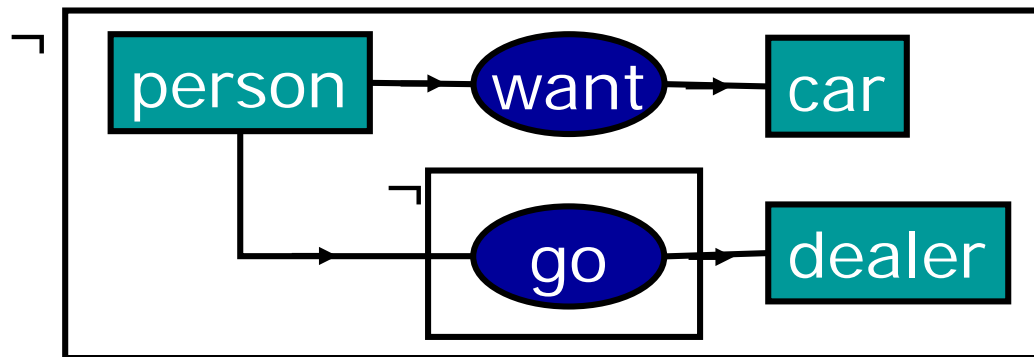
conjunctive (&) and existential (\exists) knowledge

$\forall A \forall C: \neg b(A, C)$ Ex. \neg owns(John, Sedan)

+ disjunctive (\vee) and universal (\forall) knowledge



Example: “If a person wants a car, he must go to the car dealer”



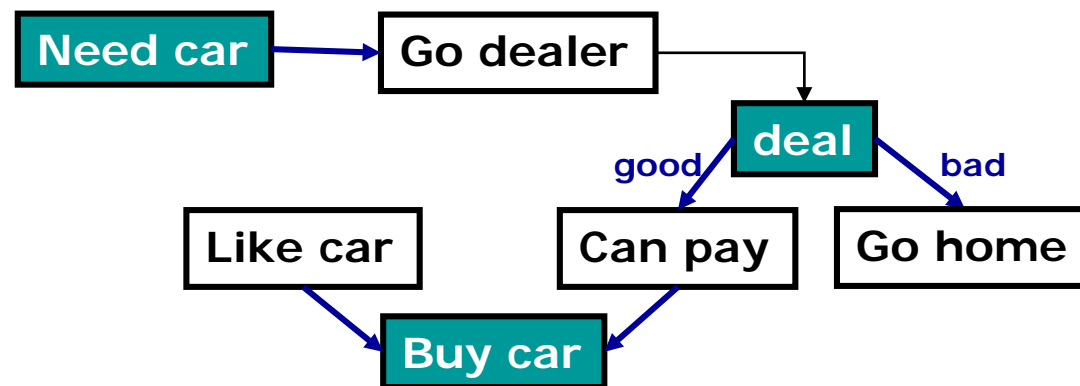
$PwC \Rightarrow PgD$
 $\neg(PwC) \vee (PgD)$
 $\neg(PwC \wedge \neg(PgD))$

Knowledge Representation: “Implicational” Semantic Networks

- Semantic network in which arcs represent logic implications.
- Sorts of “implicational” Semantic Networks:
 - *Belief or Bayesian Networks* (Judea Pearl, 1988)
 - *Causal Networks* (Chuck Riegel, 1976)
 - *Truth-Maintenance Systems, TMS* (Jon Doyle, 1979)



Example: “A person goes to a car dealer because he needs a car, and buy it if he likes the car and he can pay the price”

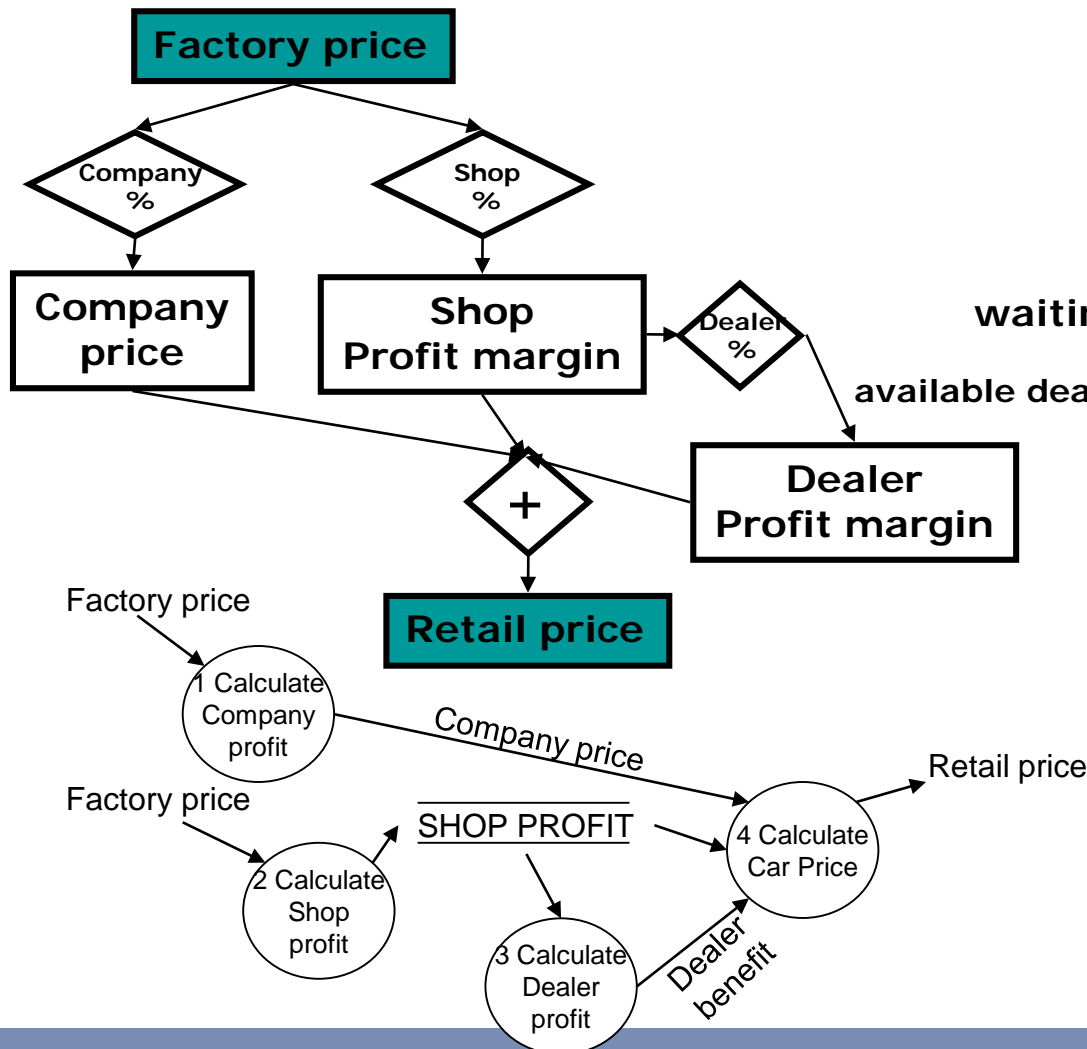


Knowledge Representation: “Executable” Semantic Networks

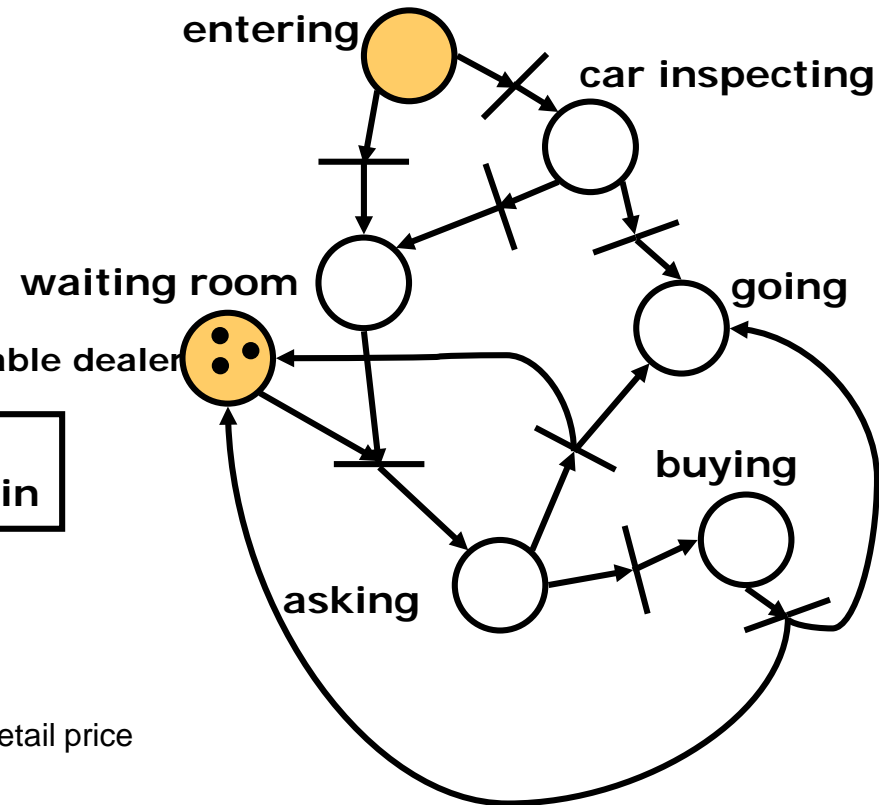
- Semantic networks that represent dynamic processes or procedural knowledge.
- General elements of the networks:
 - Message passing through the network arcs
 - Attached procedures to the network nodes
 - Graph transformations as external triggered actions
- Sorts of “executable” Semantic Networks:
 - *Dataflow diagrams*
 - *Petri Nets* (Carl Adam Petri, 1962)

“Executable” Semantic Network: examples

DFD: “Retail price calculation”



Petri Net: “Car selling”



DFD: Data Flow Diagram

Representing n-ary relations (Noy & Rector, 2006)



- **Source:** <http://www.w3.org/TR/swbp-n-aryRelations/>
- **Issues:**
 - how do we deal with cases where we need to *describe* the instances of relations, such as its certainty, strength, etc?
 - how do we represent relations among more than two individuals? ("n-ary relations")
 - how do we represent relations in which one of the participants is an ordered list of individuals rather than a single individual?
- **Examples:**
 - *Christine has breast tumor with high probability.* There is a binary relation between the person Christine and diagnosis Breast_Tumor_Christine and there is a qualitative probability value describing this relation (high).
 - *Steve has temperature, which is high, but falling.* The individual Steve has two values for two different aspects of a has_temperature relation: its magnitude is high and its trend is falling.
 - *John buys a "Lenny the Lion" book from books.example.com for \$15 as a birthday gift.* There is a relation, in which individual John, entity books.example.com and the book Lenny_the_Lion participate. This relation has other components as well such as the purpose (birthday_gift) and the amount (\$15).
 - *United Airlines flight 3177 visits the following airports: LAX, DFW, and JFK.* There is a relation between the individual flight and the three cities that it visits, LAX, DFW, JFK. Note that the order of the airports is important and indicates the order in which the flight visits these airports.

Pattern 1: Introducing a new class for a relation

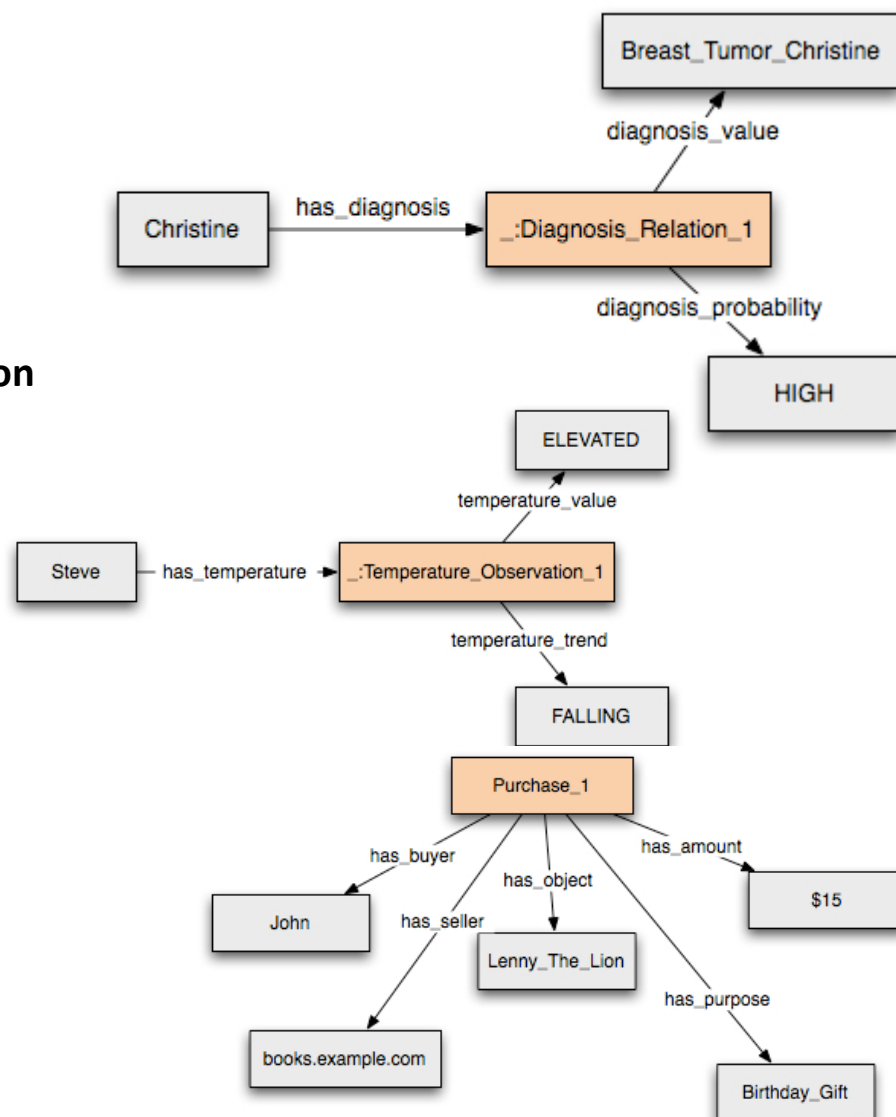
Create a new class and n new properties to represent an n -ary relation.

Useful for:

1. having additional attributes describing a relation
2. showing different aspects of the same relation
3. representing a N-ary relation with no distinguished participant

Examples:

1. *Christine has breast tumor with high probability.*
2. *Steve has temperature, which is high, but falling.*
3. *John buys a "Lenny the Lion" book from books.example.com for \$15 as a birthday gift*



Pattern 2: Using lists for arguments in a relation

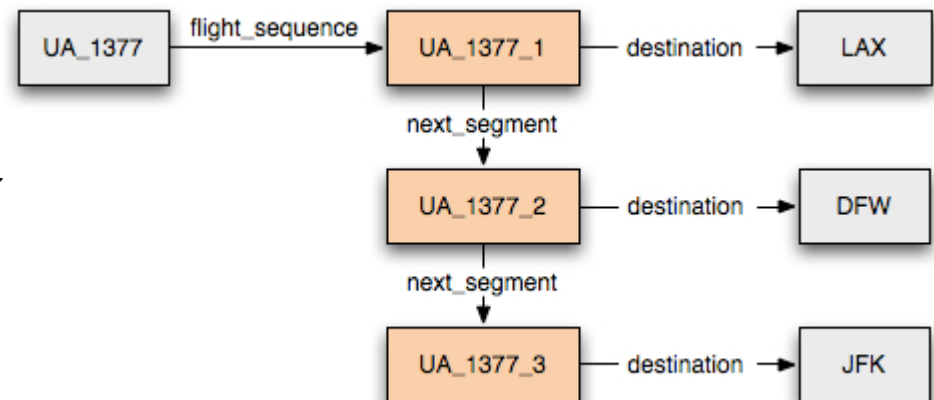
Connect these arguments into a sequence according to some relation, and to relate the one participant to this sequence (or the first element of the sequence)

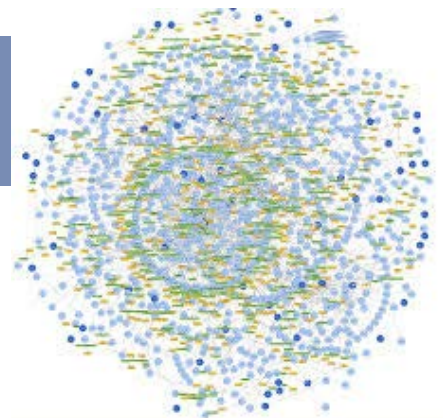
Useful for:

- representing a list or sequence of arguments

Example:

- *United Airlines flight 3177 visits the following airports: LAX, DFW, and JFK*



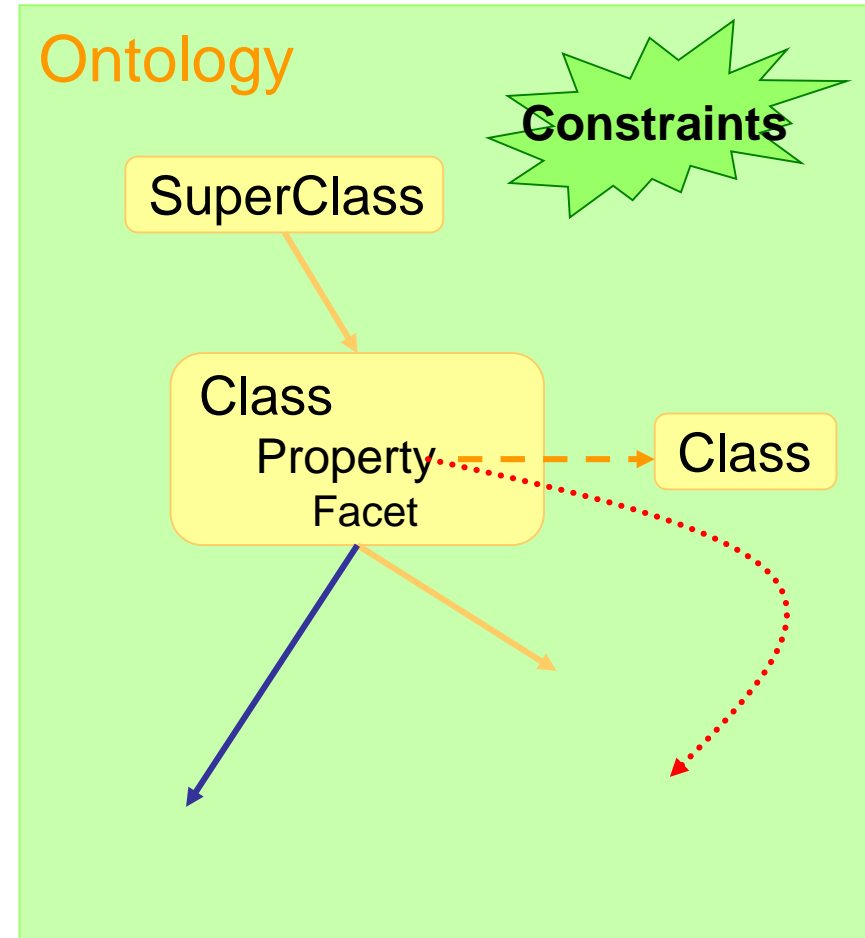


2.5. Ontologies

- Two possible definitions:
 - Ontology is an explicit specification of conceptualization.
 - Ontology is a *body of knowledge* describing some domain.
- Ontology is important for the purpose of enabling *knowledge sharing and reuse*.
- An ontology may take a variety of forms, but necessarily it will include a *vocabulary of terms*, and some *specification of their meaning*. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms.
- What does an ontology do?
 - Captures knowledge
 - Creates a shared understanding – between humans and for computers
 - Makes knowledge machine-computable
 - Makes meaning explicit – by definition and context
- Components of an ontology:
 - **Concepts**: Class of individuals
 - **Relationships** between concepts
 - *Is a kind of* relationships: they form a taxonomy
 - Other relationships: they give further structure – *is a part of*, *belongs to*, etc.
 - **Axioms**: constrains about the concepts – *Disjointness*, *covering*, *equivalence*, etc.
Ex. Family (X, Y) <- X IS_A person and Y IS_A person and (parentship(X,Y) or bothership(X,Y))
 - Individuals

Basic Elements of an Ontology

- Concepts or classes
 - Properties or slots
 - Facets
- Relationships
 - Inheritance
- Taxonomy / Network
 - Class-Subclass relationships
 - Problem specific relationships
- Constraints
- Instances



(OWL) Ontology Classes

- Class names or ***named classes***.
- Complex class expressions or **no-named classes**:

Keyword	Example	Explanation (Things that have...)
some	hasPet some Dog	... a pet that is a Dog
value	hasPet value Tibbs	... a pet that is Tibbs
only	hasPet only Cat	... pets that are only Cats.
min	hasPet min 3 Cat	... at least three pets that are Cats
max	hasPet max 5 Dog	... at most five pets that are Dogs
exactly	hasPet exactly 2 Fish	... exactly 2 Fish as pets
and	Person and (hasPet some Cat)	People that have a pet that's a Cat
or	(hasPet some Cat) or (hasPet some Dog)	... a pet that's a Cat or a Dog.
not	not (hasPet some Dog)	... do not have a pet that's a Dog

<https://www.w3.org/2007/OWL/wiki/Syntax>

Describing (OWL) ontologies: pseudocode

- **Ontology:** (ontology name elements); e.g., `(ontology animals ...)`. Elements can be: classes, properties, individuals.
- **Classes:** can be *named* and *no-named*.
 - *Named Classes:* (Class name descriptor) with descriptor another class (named or not); e.g., `class(carnivore meat-eater)`
 - *No-Named Classes:* they can be *enumerations*, set operations (*union*, *intersection*, *complement*), or *restrictions on properties*.
- **Properties:** they can be *data* and *object* properties.
 - *Data Properties:* they can contain values; `dataProperty(name domain(class) range(type))`; e.g., `dataProperty(is-dangerous domain(animal) range(Boolean))`.
 - *Object Properties:* they can contain objects, `objectProperty(name ...)`; e.g., `objectProperty (eats ...)`.
 - `domain(class): objectProperty(eats domain(animal))` “declares that property eats contain things that animals eat”.
 - `range(class): objectProperty(eats range(animal))` “declares that property eats contains things that eat animals”.
 - `inverseOf(property): objectProperty(eats inverseOf(is-eaten-by))` “declares that if A eats B then B is-eaten-by A”.
- *No-named classes:*
 - *Enumeration:* `oneOf(instance1 instance2 ...)`; e.g., `oneOf(myCat myDog myTurtle)` “my pets”.
 - *Intersection:* `intersectionOf(class1 class2 ...)`; e.g., `intersectionOf(animal female adult)` “female adult animals”.
 - *Union:* `unionOf(class1 class2 ...)`; e.g., `unionOf(animal vegetable mineral)` “all things that are animals, vegetables, or minerals”.
 - *Complement:* `complementOf(class)`; e.g., `complementOf(animal)` “all things that are not animals”.
 - *Property Restriction:* properties represent triples (**subject, predicate, object**). A property restriction refers to the subjects satisfying that restriction. It is represented as `restriction(property constraint)`, with *constraint* one of the following:
 - `someValuesFrom(class): restriction(eats someValuesFrom(animal))` “things that eat animals but maybe other things”
 - `allValuesFrom(class): restriction(eats allValuesFrom(animal))` “things that only eat animals”
 - `Cardinality(n), minCardinality(n), maxCardinality(n): restriction(eats-every-day minCardinality(2))` “things that eat two things, at least, every day”
- **Class axioms:** assertions about class *subclass*, *equivalence*, and *disjoint* relationships.
 - *Subclass:* `subclassOf(class superclass1 ...)`; e.g., `subclassOf(carnivore animal)`.
 - *Equivalence:* `equivalentClass(class1 class2)`; e.g., `equivalentClass(carnivore restriction(eats allValuesFrom(animal)))`.
 - *Disjointness:* `disjointClass(class1 class2 ...)`; e.g., `disjointClass(vegetarian carnivore)`.
 - *subproperties:* `subpropertyOf(prop1 prop2)`; e.g., `subpropertyOf(eats-veggies eats)`.
- **Individuals:** e.g., `individual(myDog type(animal) value(owner me) value(color “brown”))`

Representing ontology: pseudocode

1. Cars are vehicles
2. Cars cannot be motorcycles
3. Cars have a production year
4. Cars have components ...
5. ... at least one
6. Components belong to Cars
7. Vehicles have one and only one engine
8. Classic cars do not have security elements
9. A car is modern iff it is produced after 1999
10. Michael has a car with radio

```
1. class(Car)
   class(Vehicle)
   subClassOf(Car Vehicle)
2. disjointClass(Car Motorcycle)
3. dataProperty(prod-year domain(Car))
4. objectProperty(has domain(Car)
   range(Component))
5. subClassOf(Car restriction(has
   minCardinality(1))
6. objectProperty(belongs-to inverseOf(has))
7. objectProperty(has-engine range(Engine))
   subProperty(has-engine has)
   subClassOf(Vehicle
   restriction(has-engine cardinality(1)))
8. subClassOf(ClassicCar restriction(has
   allValuesFrom(complementOf(SecurityElement))))
8'. objectProperty(has-se range(SecurityElement))
   subClassOf(ClassicCar
   restriction(has-se cardinality(0))
9. equivalent(ModernCar intersectionOf(Car
   restriction(prod-year range(integer[> 1999])))
10. individual(MikeCar value(owner Michael)
   value(has MikeCarRadio))
   individual(MikeCarRadio type(RadioCarComponent))
   individual(Michael type(Person))
```

Protégé



- Stanford Medical Informatics, <http://protege.stanford.edu>
- Java-based standalone application. Now also available as web beta application.
- Knowledge Model:
 - Classes: concrete or abstract, hierarchy, multiple-inheritance.
 - Slots: template or own, hierarchy.
 - Facets: constraints on the slots.
 - Instances: concrete examples of classes.
 - Meta-classes: classes whose instances are classes.
 - Forms: screen layouts to edit instances of a class.
 - Queries: interface for querying the knowledge-base.
 - PAL constraints: logical constraints to represent K axioms.

Making a Car Ontology in Protégé

■ Create the hierarchy of Classes

“Cars and Motorcycles are vehicles”

“There are two sorts of cars: classic and modern”

“A car can be classic and modern, but cannot be a motorcycle”

“the components of a vehicle are mechanical (wheel and engine), electric (battery and radio), and security elements (airbag and ABS), but an engine can be also electric”

“Vehicles and Components are different things” (disjoin)

■ Create the hierarchy of Properties

Data type properties: “vehicles have a production year”

Object properties: “vehicles *have* components, and a component *is part of* a vehicle”

Property: Domain → Range

Function: “a component can only be part of a vehicle, but a vehicle can have several components (inverse is not functional)”

Transitive: “the parts of the parts of a component are parts of the component” (we need to say that components can be part of components)

■ Introduce restrictions in the relationships

Quantifiers: “vehicles have an engine” (i.e. $\text{vehicle} \sqsubseteq \exists \text{hasEngine}$) and “classic cars do not have security elements” (i.e. “all the components of a classic car are of the sort no-security-element” or $\text{component} \sqsubseteq \forall \text{has}_1 \text{SecElem}$)

Cardinality: “Vehicles must have (one or more) wheels, but cars do only have 4 and motorcycle 2”, “classic cars cannot have a production year later than 1970” (discretize allowed values of production years)

■ Necessary and sufficient condition

“A car is modern if and only if it is produced after 1999”

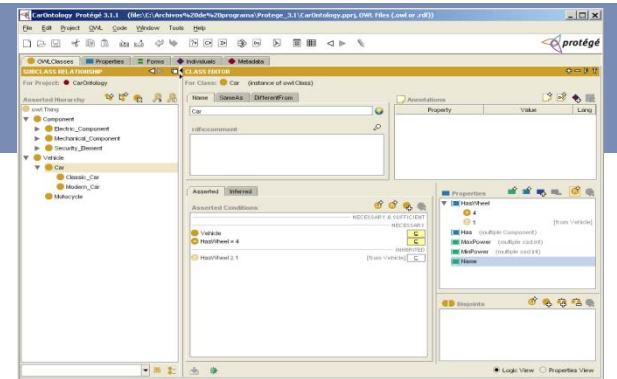
“A motorcycle is a two-wheel vehicle with an engine”

• Introduce instances

John has a modern car with an electric engine and an airbag.

Michael has a motorcycle with radio.

Mary has a car with year production 1968 and with a mechanical engine with 6 valves.



3. Knowledge Engineering

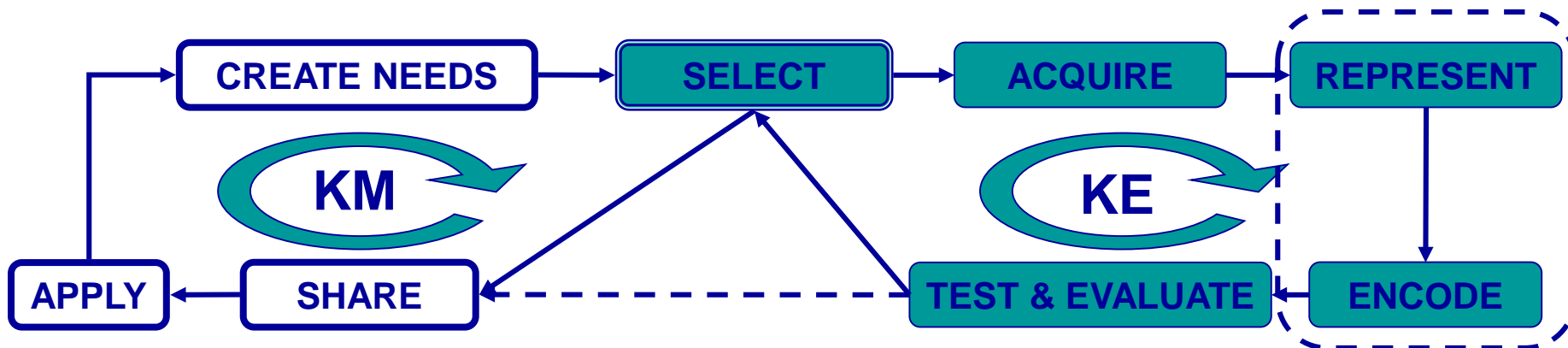
- **Definition** (*Knowledge Engineering*): part of Computer Engineering that is concerned with how knowledge is captured, stored, and validated in an agent.
- Also defined as, engineering discipline that involves **integrating knowledge into computer systems** in order to solve complex problems normally requiring a high level of human expertise.
- Also, formal methodologies for developing and maintaining knowledge-based systems.
- It is narrowly related to *Knowledge Management*, though their technologies are different.
- Some relevant topics **related** to knowledge engineering are:
 - Knowledge Life Cycle
 - Knowledge Auditing
 - Knowledge Deployment
 - Knowledge Acquisition



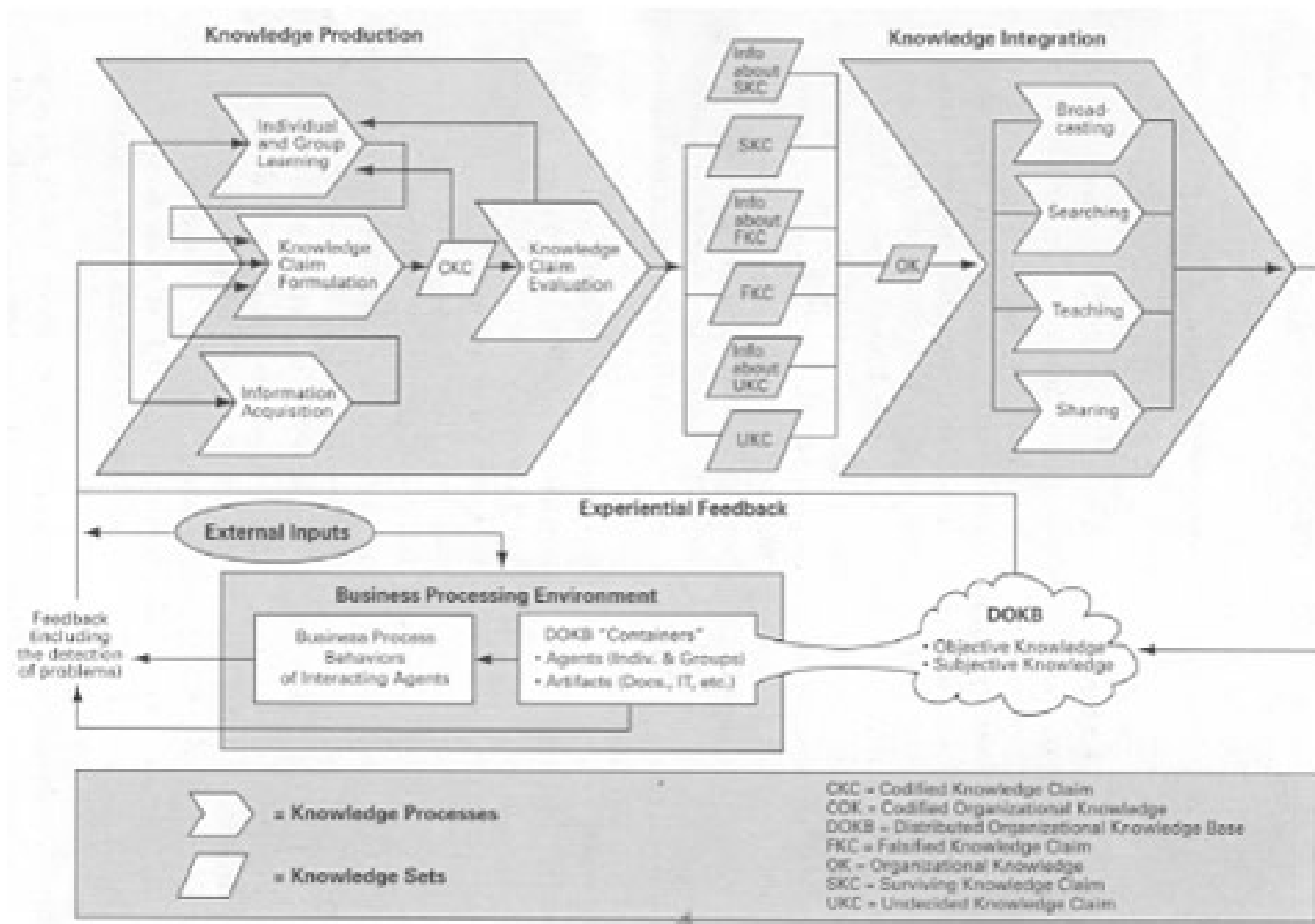
Knowledge Engineering vs. Knowledge Management



- *“In KE, knowledge is an end; in KM knowledge is a means”.*
- Both of them are based on the concept of **Knowledge Base** or Knowledge Repository.
- They are complementary: Integration of KE and KM in a global **Knowledge Life Cycle**.
 - problem selection, knowledge acquisition, knowledge representation, knowledge encoding, knowledge testing and evaluation, exploitation and maintenance.



3.1. Knowledge Lifecycle



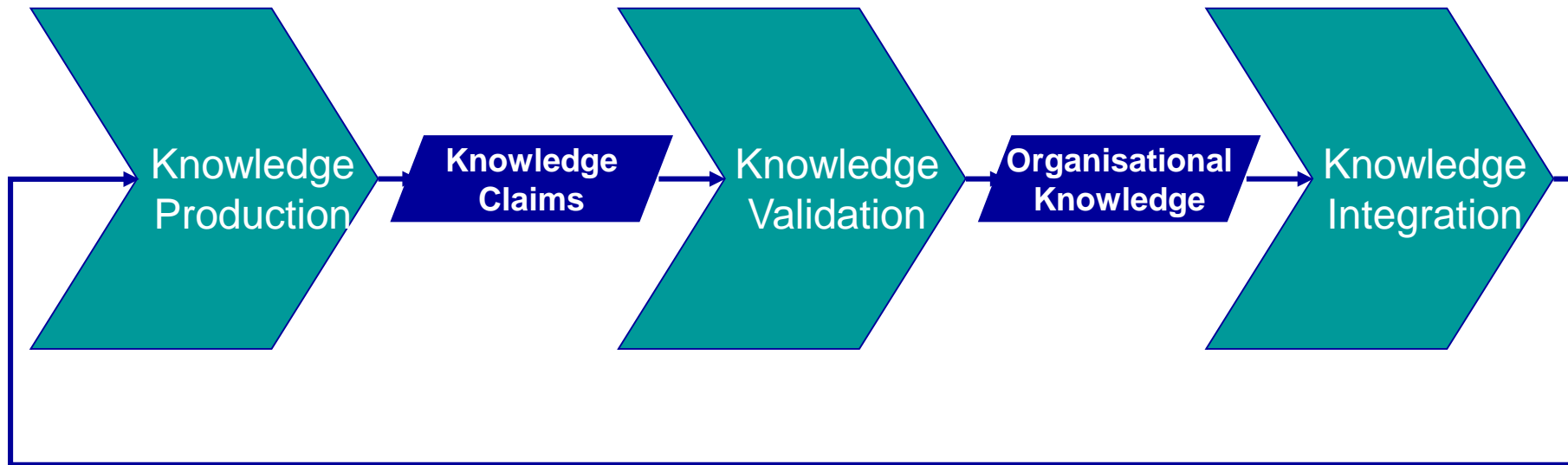
Knowledge Processes Explanation

- **Individual and group learning:** A process involving human interaction, knowledge claim formulation, and validation by which new individual and/or group knowledge is created.
- **Knowledge Claim Formulation:** A process involving human interaction by which new organizational knowledge claims are formulated.
- **Information Acquisition:** A process by which an organization either deliberately or serendipitously acquires knowledge claims or information produced by others external to the organization.
- **Knowledge Claim Evaluation (Validation):** A process by which knowledge claims are subjected to organizational criteria to determine their value and veracity.
- **Knowledge Integration:** The process by which an organization introduces new knowledge claims to its operating environment and retires old ones.

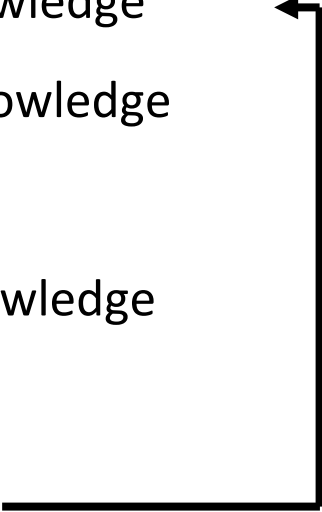
Knowledge Sets Explanation

- **Codified Knowledge Claim (CKC):** Information that has been codified, but which has not yet been subjected to organizational validation.
- **Surviving Knowledge Claim (SKC):** Codified knowledge claims that have fully satisfied an organization's validation criteria.
- **Falsified Knowledge Claim (FKC):** Codified knowledge claims whose falseness has been detected by an organization's validation criteria.
- **Undecided Knowledge Claim (UKC):** Codified knowledge claims that have not satisfied an organization's validation criteria, but which were not invalidated either. Knowledge claims requiring further study.
- **Organizational Knowledge (OK):** A complex network of knowledge and knowledge sets held by an organization, consisting of declarative and procedural rules (validated knowledge claims).

“Simplified” Knowledge Life Cycle

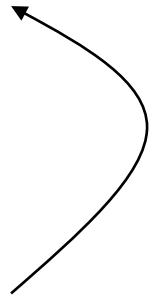


Alternative Knowledge life cycles (Liebowitz, 2000)

1. Transform Information into Kwlg.
 2. Identify & Verify Knowledge
 3. Capture & Secure Knowledge
 4. Organize Knowledge
 5. Retrieve & Apply Knowledge
 6. Combine Knowledge
 7. Create Knowledge
 8. Learn Knowledge
 9. Distribute/Sell Knowledge
- 

Alternative Knowledge life cycles (Liebowitz&Beckman, 2000)

STAGE 1: IDENTIFY
STAGE 2: CAPTURE
STAGE 3: SELECT
STAGE 4: STORE
STAGE 5: SHARE
STAGE 6: APPLY
STAGE 7: CREATE
STAGE 8: SELL



Identify is to determine competencies, sourcing strategy, and knowledge domains.

Capture the existing knowledge is formalized during this phase.

Select consists on assessing knowledge relevance, value and accuracy, and resolve conflicting knowledge.

Store: The knowledge is stored by representing the corporate memory in a knowledge repository with various knowledge schema.

Share & Apply: Then, the stored knowledge can be shared and finally *applied* in making decisions, solving problems, automating or supporting work, job aids, and training.

Create: New knowledge can be discovered (with or without the use of the previous one) through research, experimenting, and creative thinking.

Sell: Apart of applying the knowledge in stage 6, it can be also sell. That's to say, new knowledge-based products and services can be developed and marketed.

Alternative Knowledge life cycles (Marquardt, 1996), ...

(Marquardt, 1996)

1. Acquisition
2. Creation
3. Transfer and utilization
4. Storage

(Spek & Spijkervet, 1997)

1. Developing new knowledge
2. Securing new & existing K.
3. Distributing Knowledge
4. Combining available K.

(Ruggles, 1997)

1. Generation: Creation, Acquisition, Synthesis, Fusion, Adaptation
2. Codification: Capture, Representation
3. Transfer

(Holsapple & Joshi, 1997)

1. Acquiring Knowledge: Extracting, Interpreting, Transferring
2. Selecting Knowledge: Locating, Retrieving, Transferring
3. Internalizing Knowledge: Assessing, Targeting, Depositing
4. Using Knowledge
5. Generating Knowledge: Monitoring, Evaluating, Producing, Transferring
6. Externalizing Knowledge: Targeting, Producing, Transferring

(Wiig, 1993)

1. Creation and Sourcing
2. Compilation and Transformation
3. Dissemination
4. Application and Value realization

(O'Dell, 1996)

1. Identify
2. Collect
3. Adapt
4. Organize
5. Apply
6. Share
7. Create

(Dataware Technologies, 1998)

1. Identify the (business) problem
2. Prepare for change
3. Create the KM team
4. Perform the knowledge audit and analysis
5. Define the key features of the solution
6. Implement the building blocks for KM
7. Link knowledge to people

(Van der Spek & Hoog, 1998)

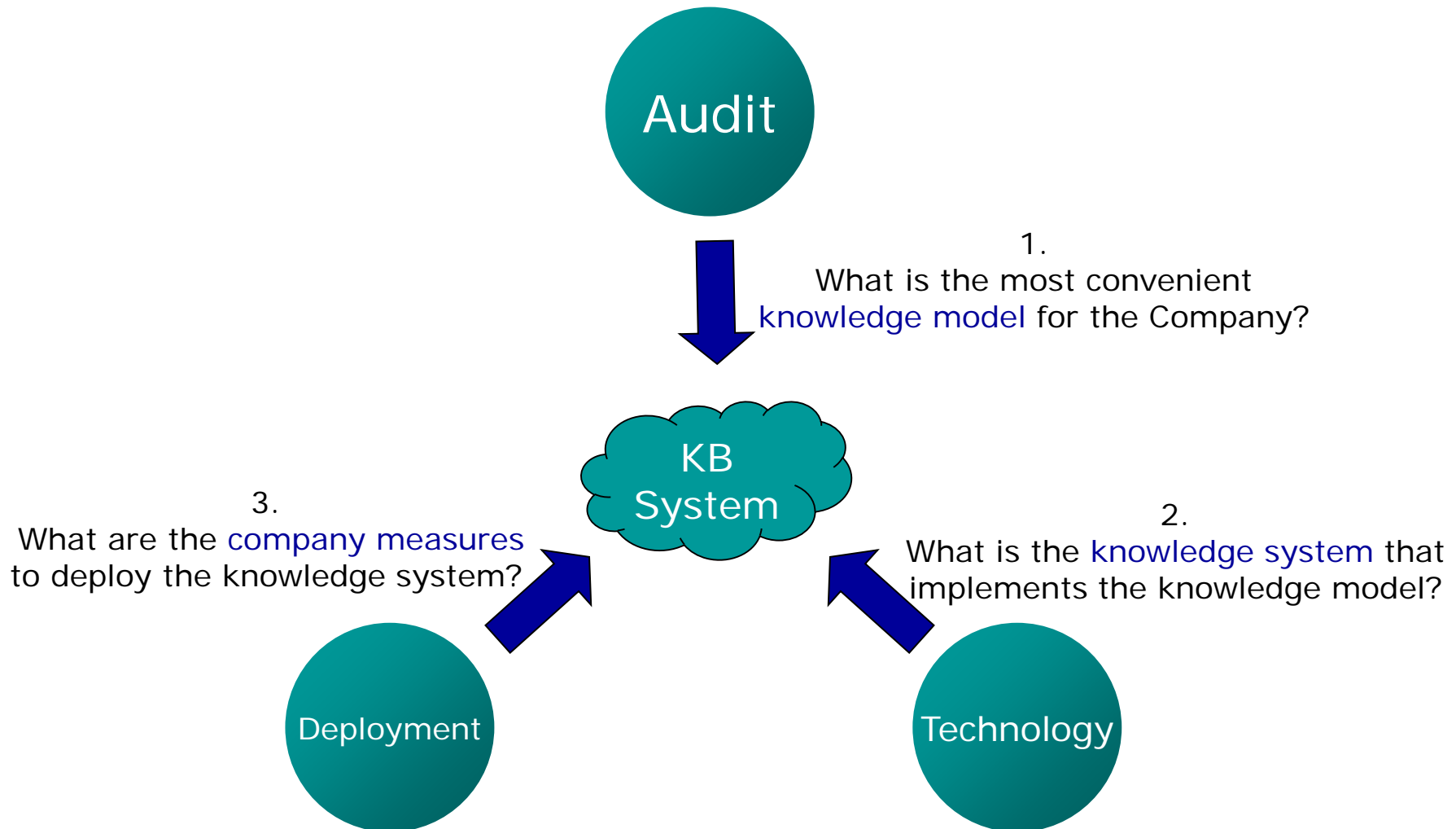
1. Conceptualize: Make inventory of existing K., Analyze strong and weak points
2. Reflect: Decide on required improvements, Make plans to improve
3. Act: Secure, Combine, Distribute, and Develop knowledge
4. Review: Compare old and new situation, Evaluate achieved results

SGKM: Knowledge Processes

- *Knowledge processes*: any of the processes involved in the knowledge life cycle.
- *Knowledge processing*: act of applying some knowledge process (ex. knowledge production or integration).
- *Knowledge Management* is about an action that seeks to have an impact on knowledge processing (ex. to design a portal to enhance knowledge sharing).



The three components of a knowledge-oriented project



3.2. Knowledge Auditing

Auditing knowledge for a particular target area consists on:

*“identifying which knowledge is needed and **available** for that area, which knowledge is **missing**, who has the knowledge (**source**), and how it is being used (**destination**)”.*

1. Identify the currently existing knowledge in the targeted area:

Determine existing and potential sinks, sources, flows, and constraints.

Identify and locate explicit and tacit knowledge.

Build a **knowledge map** of the taxonomy and flow of knowledge.

2. Identify the currently missing knowledge in the targeted area:

Perform gap analysis to detect missing knowledge.

Determine who needs the missing knowledge.

3. Provide recommendations

Knowledge Auditing Algorithm

- What are the categories of knowledge in the targeted area?
- Which of them are currently available?
- If available,
 - How this knowledge is used?
 - What are the sources of this knowledge?
 - Who is using this knowledge? How often?
 - Who are new potential users of this knowledge?
 - What's the process or processes to obtain that knowledge?
 - How is this knowledge adding value or benefit?
 - What influences this knowledge?
 - What are the elements that identify, use, or transform this knowledge?
 - How is this knowledge delivered from? Are there other delivering alternatives?
 - Who are the experts (in the company) in this sort of knowledge?
- What categories of knowledge do you need to do your work better?
- For all of them,
 - How much your work can be improved from it?
 - What are the potential sources of this knowledge?
 - What are your unanswered questions? For each one,
 - What is the sort of knowledge missed?
 - Which departments/people could answer these open questions?
 - Which departments/people are looking for similar answers? For each one,
 - What level in the organization this department/person has?
 - If a person, how old is this person in the company?
 - Why did they ask these questions similar to yours?
 - Is someone in the organization putting barriers to this sort of knowledge orientation?
 - What are the main reasons to make errors/mistakes concerning this knowledge?

3.3. Knowledge Deployment: actions

1. Obtain management buy-in (acceptance & support). (J. Liebowitz, 2000)
2. Survey and map the knowledge landscape.
3. Plan the knowledge strategy.
4. Create/define K-related alternatives and potential initiatives.
5. Portray benefit expectations for KM initiatives.
6. Set KM priorities.
7. Determine key knowledge requirements.
8. Acquire key knowledge.
9. Create integrated knowledge transfer programs.
10. Transform, distribute, and apply knowledge assets.
11. Establish and update KM infrastructure.
12. Make knowledge assets.
13. Construct incentive programs.
14. Coordinate KM activities and functions enterprise-wide.
15. Facilitate knowledge-focused management.
16. Monitor Knowledge Management.



time

An 8-step agenda

(J. Liebowitz, 2000)

1. Develop a broad **vision** of the KM practice and obtain management **buy-in**.
2. Pursue targeted KM focus.
3. Allow **team** members to focus full time on KM and build KM professional teams.
4. Install KM impact and benefit **evaluation** methods.
5. Implement **incentives** to manage knowledge.
6. **Teach** metaknowledge to everyone.
7. **Ascertain** that implemented KM activities provide opportunities, capabilities, motivations, and permissions for individuals and the enterprise to act intelligently.
8. Create **supporting infrastructure**.



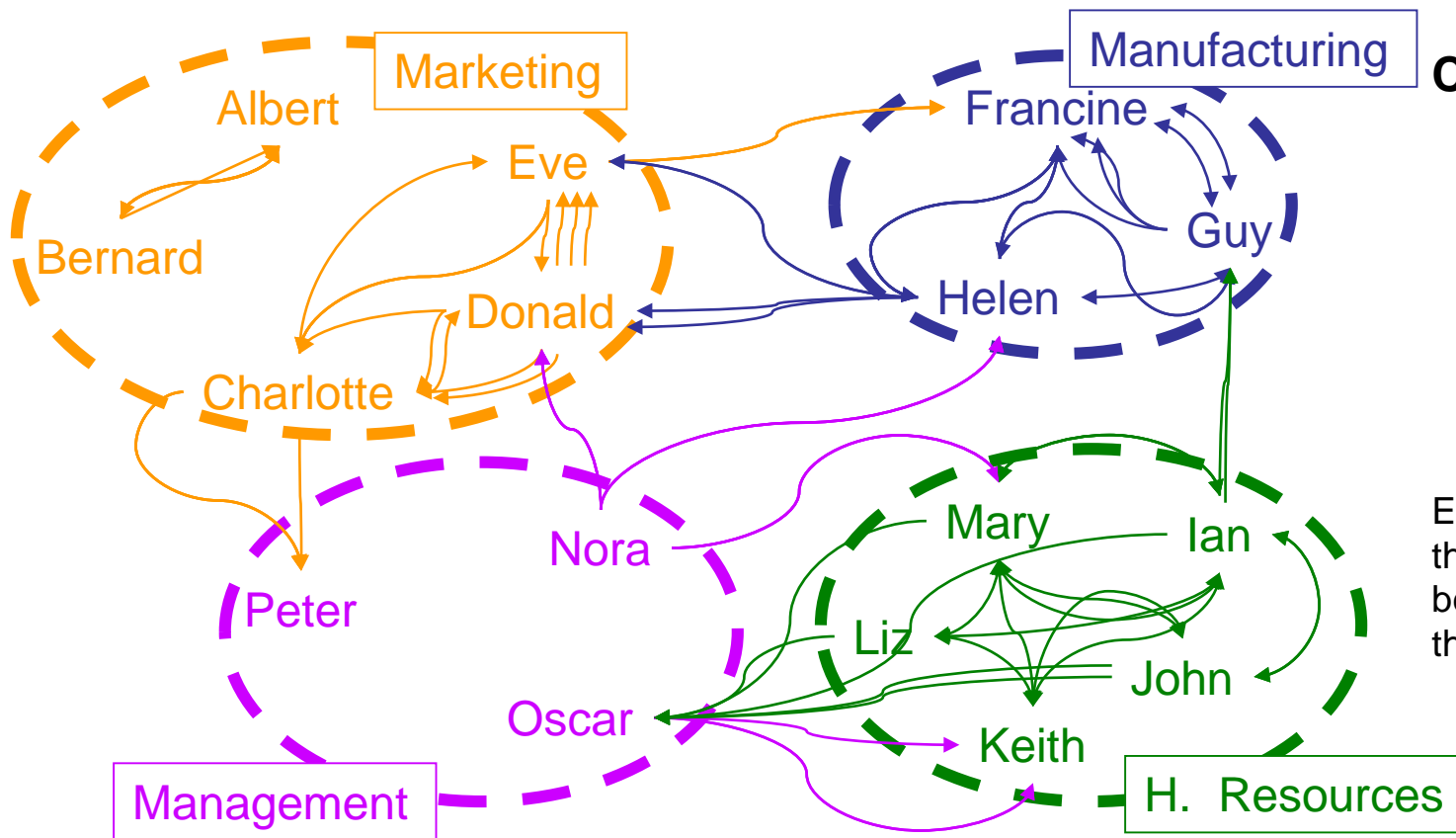
relevance

Knowledge Maps

- Representation of the knowledge inside the company
- Purposes:
 - Generate ideas
 - Design a complex structure
 - Communicate complex ideas
 - Aid learning by integrating new and old knowledge
 - Assess understanding
 - Diagnose misunderstanding
- Sorts of knowledge maps:
 - Organizational Maps
 - Expertise Maps
 - Concept Maps

Organizational Maps

They are used to show the interactions between company members.



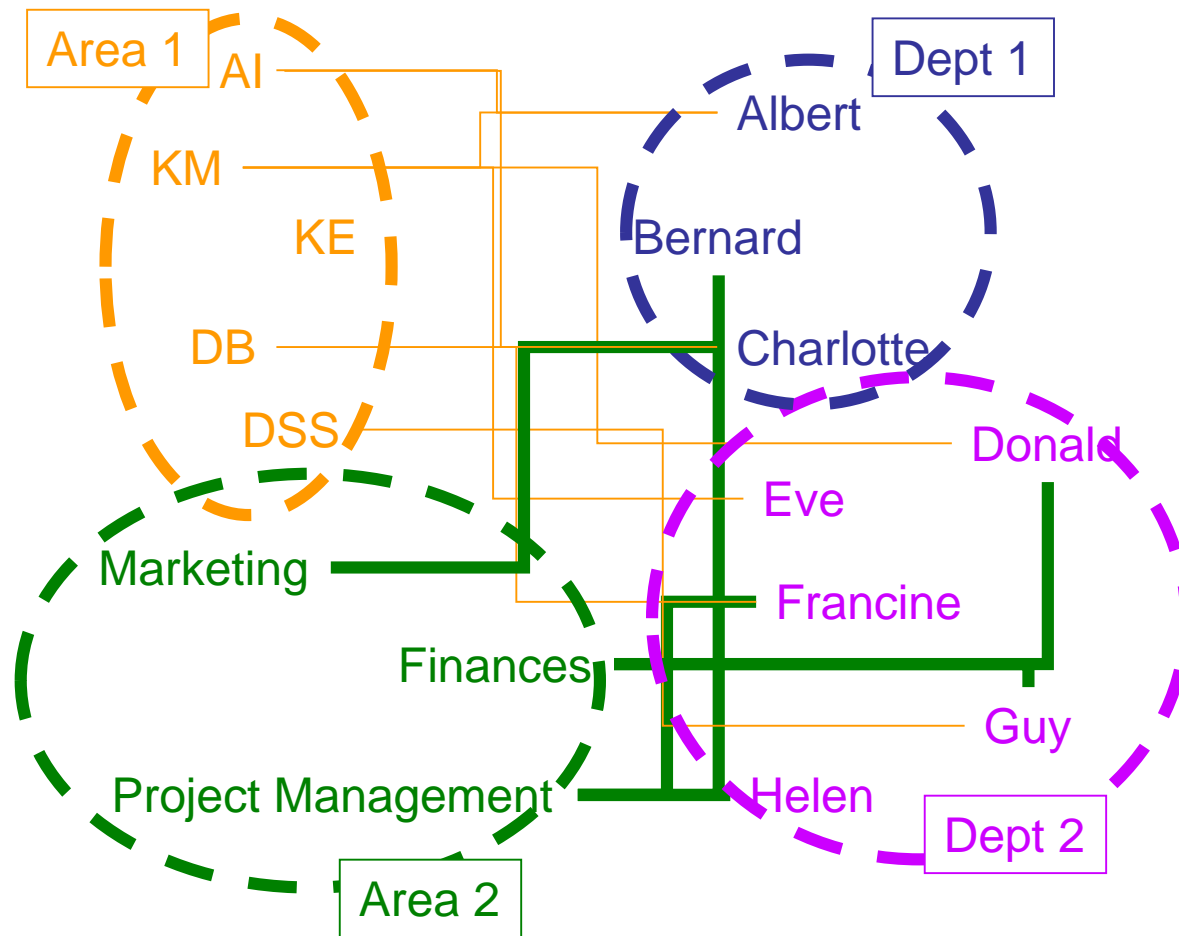
Collaborations:

Close
Distant
Isolations
Unidirectional
Hierarchies
Etc.

Ex: by the analysis of the emails/internal calls between members in the company.

Expertise Maps

They are used to show who knows things in the company.



Expertise:

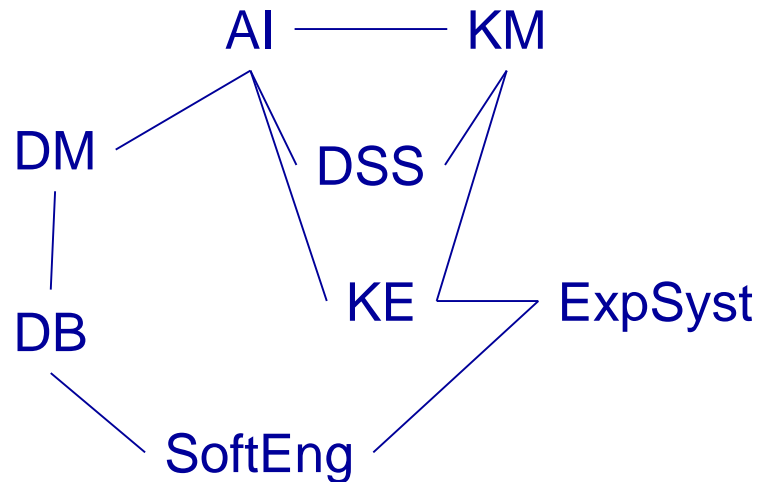
Someone/nobody
Who/What dept.
Working team
People selection
Employee formation
Etc.

Ex: by the analysis of people participating in projects, papers, Reports, meetings, etc. and Their role/responsibility in that activities.

Concept Maps

They are used to know the relationships between company concepts.

Concepts can be: objects, resources, products, etc.



Semantic Knowledge Maps: TCR Knowledge Maps

(Newbern & Dansereau, 1993)

- The links in the Map have a **meaning**.

- Meanings:

- Descriptive Links

C – characteristic

P – part of

T – type or subconcept of

- Dynamic Links

I – influences

L – leads to

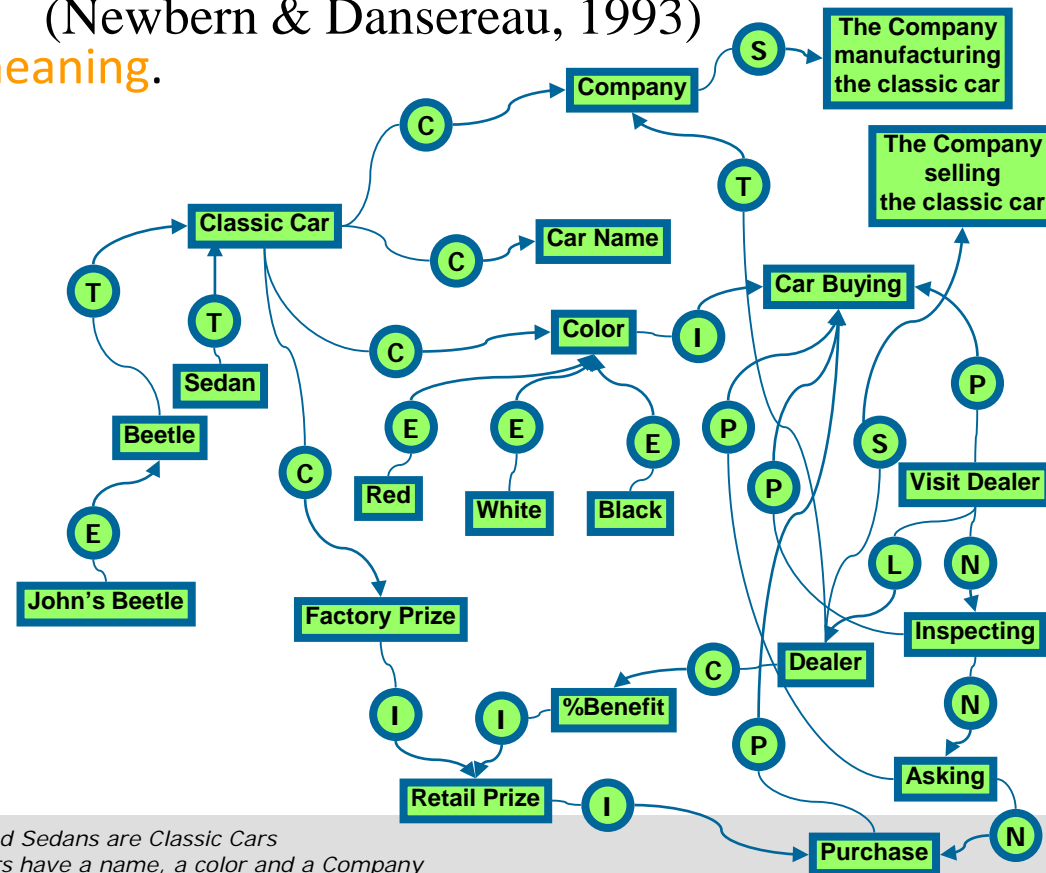
N – next

- Instructional Links

A – analogy

S – side remark

E – example



- (1) Beetles and Sedans are Classic Cars
- (2) Classic cars have a name, a color and a Company
- (3) John has a Beetle
- (4) Classic cars can be red, white, and black
- (5) The color of the car is relevant to buy it
- (6) Cars have a factory prize which is the basis to calculate the retail price
- (7) Red cars are more expensive than the rest of cars
- (8) Buying a car consists of going to a dealer, inspecting a car, asking for car properties and prize, and buy it.
- (9) Car dealers fix a % of benefit on the cars (which conditions the retail prize or final prize)
- (10) Buying a car depends on the prize

Newbern D., Dansereau D.F.: Knowledge Maps for Knowledge Management (chap 9). In: Knowledge Management Methods, Wiig K.M, 1995.

Constructing Semantic Knowledge Maps

(Liebowitz , 2001)

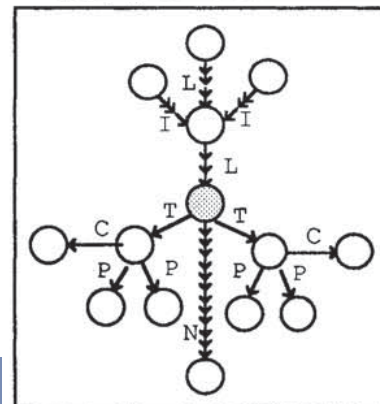
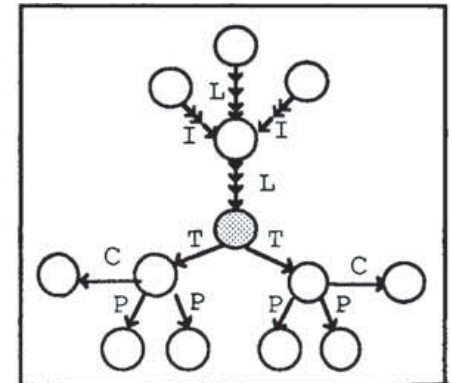
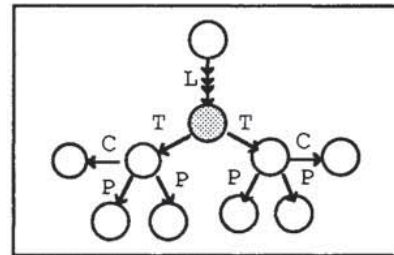
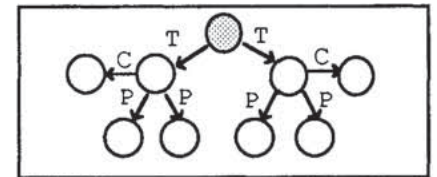
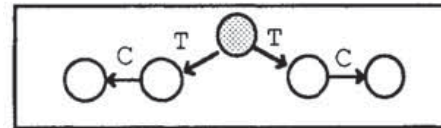
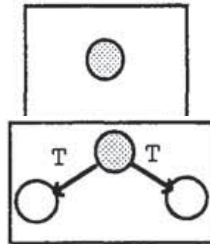
1. Make a list of important concepts or main ideas.
2. For each concept or idea,
 - 2.1. Add a node in the map, labeled with the concept.
 - 2.2. Ask the following questions and draw links on the map,
 - 2.2.1. Can this concept be broken down into sub concepts (T-link)?
 - 2.2.2. For each sub concept or concept type,
 - 2.2.2.1. What are the features of that type (C-link)?
 - 2.2.2.2. What are the important parts of that type (P-link)?
 - 2.2.2.3. For each part, what are the features (C-link)?
 - 2.2.3. What led to a starting node (L-link)?
 - 2.2.4. What does a starting node lead to (L-link)?
 - 2.2.5. Which things influence a starting node (I-link)?
 - 2.2.6. What does a starting node influence (I-link)?
 - 2.2.7. What happens next (N-link)?
 - 2.2.8. Does anything require an analogy, remark or example (A,S,E-links)?
3. Review the map

Jay Liebowitz. Knowledge Management: Learning from Knowledge Engineering. CRC Press 2001.

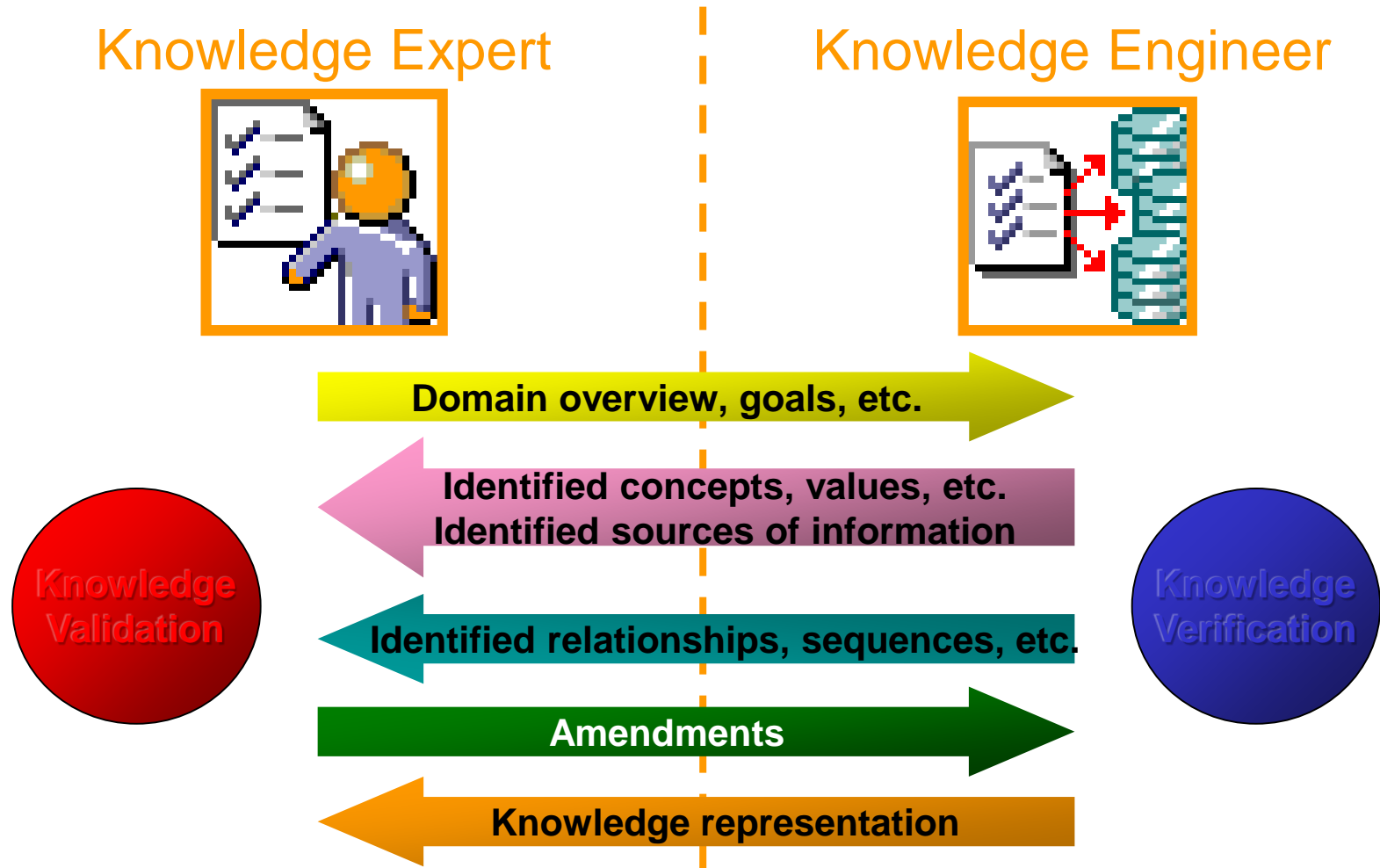
Graphical Example

■ For each concept

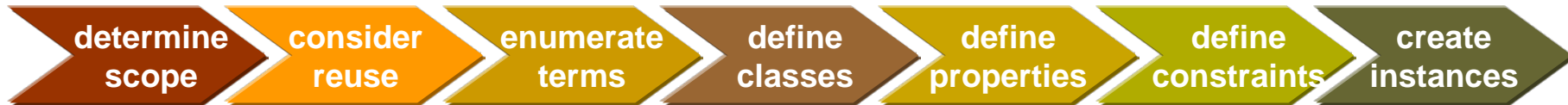
- Find types
- Find characteristics of types
- ... and their important parts
- ... and characteristics of these parts
- Leading to starting node
- Starting node leads to ...
- Influences of starting node
- Starting node influences ...
- What happens next
- Analogies and examples



3.4. Knowledge Acquisition



3.5. Knowledge Development Technology: 101



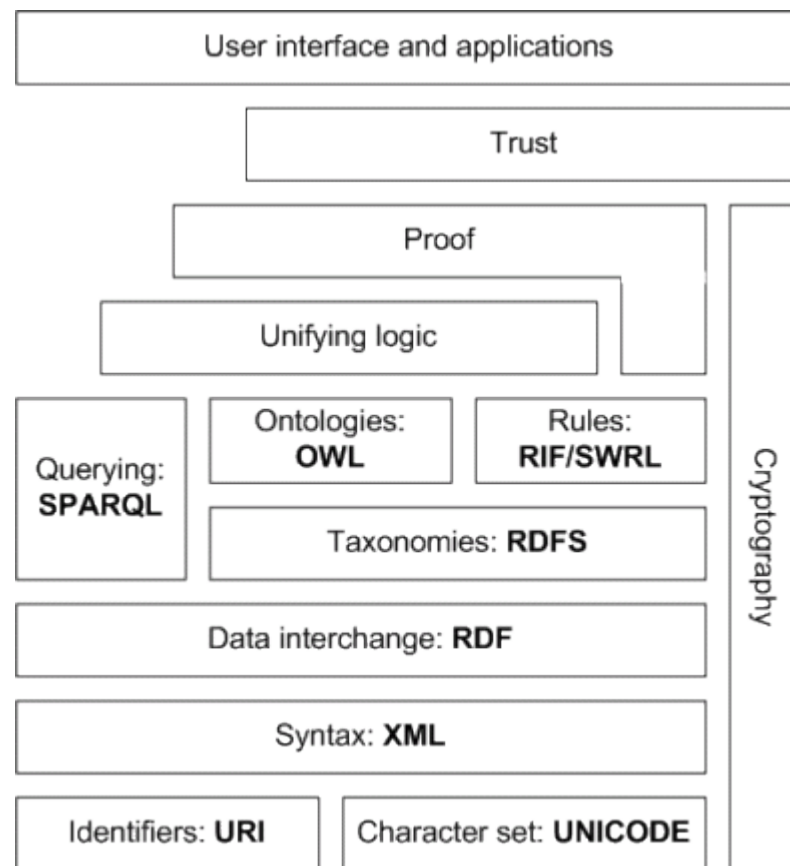
- **Determine Domain and Scope**
 - What is the domain that the ontology will cover?
 - For what we are going to use the ontology?
 - For what types of questions the information in the ontology should provide answers (competency questions)?
- **Consider reuse**
 - to save the effort
 - to interact with the tools that use other ontologies
 - to use ontologies that have been validated through use in applications
- **Enumerate important terms**
 - What are the terms we need to talk about?
 - What are the properties of these terms?
 - What do we want to say about the terms?
- **Define Classes**
 - A class is a concept in the domain
 - A class is a collection of elements with similar properties
 - Class inheritance
 - Classes usually constitute a taxonomic hierarchy (a subclass-superclass hierarchy)
 - A class hierarchy is usually an IS-A hierarchy.
 - If you think of a class as a set of elements, a subclass is a subset
- **Define Properties (Slots)**
 - Slots in a class definition describe attributes of instances of the class and relations to other instances
 - Types of properties
 - “intrinsic” properties: flavor and color of wine
 - “extrinsic” properties: name and price of wine
 - parts: ingredients in a dish
 - relations to other objects: producer of wine (winery)
 - Simple and complex properties
 - data properties (attributes): contain primitive values (strings, numbers)
 - object properties: contain (or point to) other objects (e.g., a winery instance)
 - A subclass inherits all the properties from the superclass
 - If a class has multiple superclasses, it inherits properties from all of them
- **Define Constraints (Facets)**
 - Property constraints (facets) describe or limit the set of possible values for a property
 - cardinality – the number of values a property has
 - value type – the type of values a property has
 - Minimum and maximum value – a range of values for a numeric prop.
 - Default value
 - Domain of a property
 - Range of a property
- **Create Instances**
 - Create an instance of a class
 - Assign property values for the instance frame

Conclusions

- What is knowledge representation?
- The importance of representing knowledge
- Differentiate btw data, information, and knowledge
- Distinguish between the basic sorts of knowledge:
 - Implicit vs. explicit
 - declarative vs. procedural
 - individual, group, organizational
 - informal, semi-structured, structured, formal
- Recognize the importance of knowledge semantics
- Being able to manage five models to formalize knowledge:
 - First order logic
 - Rules and production systems
 - Frames and Scripts
 - Semantic networks
 - Ontologies
- Practice with three languages/tools to represent knowledge:
 - CLIPS for rules
 - COOL: CLIPS for frames (object oriented)
 - Protégé for ontologies
- Know how to represent complex relationships
 - Taxonomical (hierarchical), mereological (part-of), generic (object properties).
 - Unary, binary, N-ary
- What is knowledge engineering?
- The utility of knowledge engineering methods
- Distinguish btw knowledge engineering and k. management
- The knowledge life cycle:
 - what is it? What's its purpose?
 - The main loop and parts (knowledge production, validation, and integration).
- What are the relevant actions in a knowledge oriented project
- Knowledge auditing:
 - what are the constituting Important steps
 - how to elaborate a document/Report (important sections)
 - Method 1: knowledge auditing procedure
- Knowledge deployment:
 - What is it?
 - Method 2: sequence of actions for knowledge deployment
 - Method 3: simplified 8-step agenda
- Knowledge maps: meaning, purpose, and uses
 - Organizational knowledge maps: defining, and using
 - Expertise knowledge maps: defining, and using
 - Concept knowledge maps: defining, and using
 - Semantic knowledge maps
 - Method 4: constructing a semantic knowledge map
- What is knowledge acquisition? Steps of KA
- Method 5: ontology development with 101

4. Knowledge Representation in the Web

- The W3C
- The Web:
 - HTML
- The Semantic Web:
 - XML
 - RDF
 - RDFS
 - **OWL**
 - SPARQL
 - RIF/SWRL



The World Wide Web Consortium

- The World Wide Web Consortium (W3C) is an international consortium where member organizations, a full-time staff, and the public work together to develop Web standards.
- W3C's mission is to lead the World Wide Web to its full potential by developing protocols and guidelines that ensure long-term growth for the Web.
- <http://www.w3c.org>
- Tutorials on [W3C standards and products](http://www.w3schools.com) (www.w3schools.com)



HTML, XHTML and DHTML

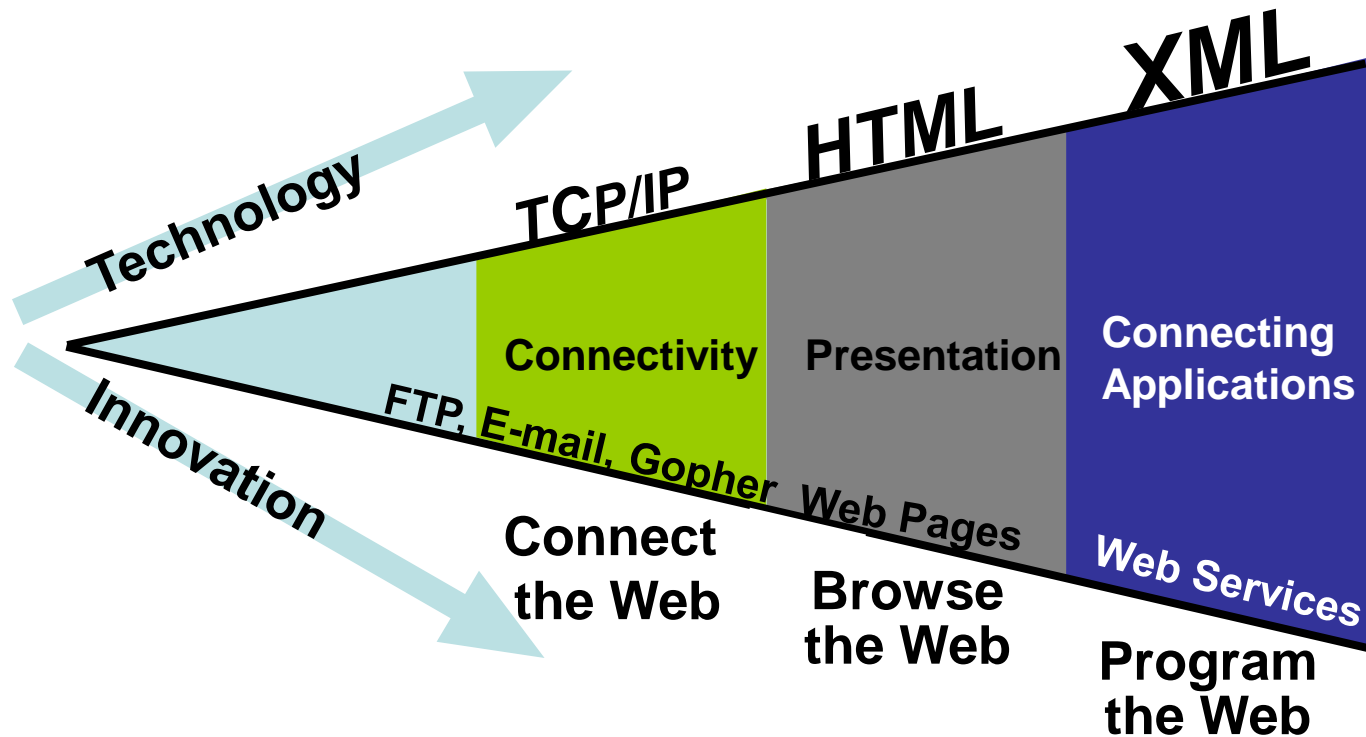


- HTML was designed for formatting text on a Web page.
- HTML **limitations**:
 - Cannot deal with the content of a Web page.
 - Cannot be used to describe or to catalog data in the web.
 - It is not extensible.
 - “Standard” representation but browser-dependent appearance.
- eXtensible HyperText Markup Language (XHTML): getting stricter and cleaner:
 - XHTML elements must be **properly nested**
 - XHTML elements must always be **closed**
 - XHTML elements must be in **lowercase**
 - XHTML documents must have **one root element**
 - Attribute names must be in **lower case**
 - Attribute values must be **quoted**
 - Attribute minimization is **forbidden**
 - The id attribute **replaces** the name attribute
 - The XHTML DTD defines **mandatory** elements
- Dynamic HTML (DHTML): HTML + Style Sheets + JavaScript.

XML: eXtensible Markup Language



- XML specifies the structure and content of a document.
- **Extensible**: tags are defined by users as required.
- **Markup**: tags mark data with meaning (information).
- XML is to structure, store and to send **information**.



XML Outlines

■ Main features:

- markup language
- ... to describe data
- ... by tags that are not predefined (you define your own tags)
- ... that uses a DTD or a Schema to describe the data
- ... and self-descriptive.

■ XML reserved symbols: &, <, >, ' , " , and ;.

■ Basic elements:

- Tags: `<tag-name> content </tag-name>`
- Attributes: `<tag-name att-name="content">`
- Comments: `<!-- content -->`
- Tag content can be text and/or new embedded tags

■ HTML browsers supporting XML:

- Microsoft Internet Explorer ≥5.0
- Netscape Navigator ≥ 6 (option “View Page Source”)
- Firefox

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<sell>
  <client>
    <name>John Smith</name>
    <address>1400 Smith St.</address>
    <telephone>6666666666</telephone>
    <profession>Architect</profession>
  </client>
  <car>
    <name>Beetle</name>
    <license_plate>XXXXXX</license_plate>
  </car>
  <price>$15000</price>
</sell>
```

standalone="no": DTD/schema in file apart.
 standalone="yes": DTD/schema within xml file.
 without standalone ATT: no DTD/schema.

XML Management

- Document Type Definition or Schema (**DTD, Schema**): XML document that defines the content structure of other XML documents.
- XML Path Language (**XPath, XQuery**): locates information in XML documents.
- eXtensible Stylesheet Language (**XSL**): XSL **describes** how the XML document should be displayed. It transforms XML into HTML before it is displayed by the browser.

DTD outlines



- <http://www.w3schools.com/dtd/>

- Tags are declared and hierarchically related as ELEMENT

```
<!ELEMENT tagname (listofsubelements)>
```

- Tag attributes are declared with ATTLIST

```
<!ATTLIST tagname attnamei atttypei defaultvaluei>
```

- The simplest element content is text

```
<!ELEMENT tagname (#PCDATA)>
```

- Subelements can be exactly 1, 0/1 (?), 0/N (*), or 1/N (+)

```
<!ELEMENT tagname (e1,e2?,e3*,e4+)>
```

(ex. e1=name, e2=wife'sname, e3=siblingname, e4=address)

- Subelements can be alternatives

```
<!ELEMENT tagname (e1|e2|e3)>
```

- Attribute types can be

CDATA	text
(value1 value2 ... valueN)	attribute values
ID	unique identifier
IDREF	id of another element
IDREFS	list of other ids

- Attribute default values can be

value	default value
#REQUIRED	the attribute is required
#IMPLIED	the attribute is optional
#FIXED value	the attribute value is forced to be value

See some examples at http://zvon.org/xxl/DTDTutorial/General_spa/book.html

DTD-XML car's example

```

<!ELEMENT oldcarsworld (company*,car+)>
<!ELEMENT company (country?)>
<!ATTLIST company name ID #REQUIRED>
<!ELEMENT country (#PCDATA)>
<!ELEMENT car
  (model?,horsepower,production+,
   color*,price,wheels)
>
<!ATTLIST car
  name (BEETLE|SEDAN|JEEP|TOPOLINO) #REQUIRED
  company IDREF #REQUIRED>
<!ELEMENT model (#PCDATA)>
<!ELEMENT horsepower (HP|(HPmin?,HPmax?))>
<!ELEMENT HP (#PCDATA)>
<!ELEMENT HPmin (#PCDATA)>
<!ELEMENT HPmax (#PCDATA)>
<!ELEMENT production (start?,finish?)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT finish (#PCDATA)>
<!ELEMENT price (factory_price,retail?)>
<!ELEMENT color EMPTY>
<!ATTLIST color name (R|W|B|Y|DARK|OTHER)
  #REQUIRED>
<!ELEMENT factory_price (#PCDATA)>
<!ELEMENT retail_price (#PCDATA)>
<!ELEMENT wheels (#PCDATA)>

```



```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE oldcarsworld SYSTEM "cars.dtd">

<oldcarsworld>
  <company name="VolsWagen">
    <country>Germany</country>
  </company>

  <company name="Fiat">
    <country>Italy</country>
  </company>

  <car name="BEETLE" company="VolsWagen">
    <model>1500</model>
    <horsepower>
      <HPmin>50</HPmin> <HPmax>90</HPmax>
    </horsepower>
    <production>
      <start>1938</start> <finish>1989</finish>
    </production>
    <production>
      <start>2000</start>
    </production>
    <color name="B"></color>
    <price>
      <factory_price>8000 Euro</factory_price>
    </price>
    <wheels>4</wheels>
  </car>

  <car name="TOPOLINO" company="Fiat">
    <model>500</model>
    <horsepower>
      <HP>50</HP>
    </horsepower>
    <production>
      <start>1936</start> <finish>1955</finish>
    </production>
    <color name="R"/> <color name="W"/> <color name="B"/>
    <price>
      <factory_price>8000€</factory_price>
    </price>
    <wheels>4</wheels>
  </car>
</oldcarsworld>

```

XML Schema outlines

- An XML Schema (extension .XSD) defines:
 - elements that can appear in a document
 - attributes that can appear in a document
 - which elements are child elements
 - the order of child elements
 - the number of child elements
 - whether an element is empty or can include text
 - data types for elements and attributes
 - default and fixed values for elements and attributes
- Elements: `<xs:element name="xxx" type="yyy"/>`
 - `xs:string`
 - `xs:decimal`
 - `xs:integer`
 - `xs:boolean`
 - `xs:date`
 - `xs:time`
- Default values: `<xs:element ... default="red"/>`
- Fixed values: `<xs:element ... fixed="red"/>`
- Attributes: `<xs:attribute name="xxx" type="yyy"/>`
- Default att values: `<xs:attribute ... default="yyy"/>`
- Fixed att values: `<xs:attribute ... fixed="yyy"/>`
- Required atts: `<xs:attribute ... use="required"/>`
- More complex aspects:
 - Restriction on values
 - Complex Elements (empty, elements-only, text-only, mixed)
 - Indicators (all, choice, sequence / maxOccurs, minOccurs / ...)
 - The ANY element (any expression inside)

```

XML -----
<?xml version="1.0"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me!</body>
</note>

DTD -----
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>

Schema -----
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">

  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"
maxOccurs="unbounded"/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>

```

XSD – Nested Style

```
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" version="1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="oldcarsworld">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="company"
          maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="country" type="xsd:string"
                minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:ID"
              use="required"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="car" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="model" type="xsd:string"
                minOccurs="0"/>
              <xsd:element name="horsepower">
                <xsd:complexType>
                  <xsd:choice>
                    <xsd:element name="HP" type="xsd:int"/>
                    <xsd:sequence>
                      <xsd:element name="HPmin" type="xsd:int"
                        minOccurs="0"/>
                      <xsd:element name="HPmax" type="xsd:int"
                        minOccurs="0"/>
                    </xsd:sequence>
                  </xsd:choice>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="production"
                maxOccurs="unbounded">
                  <xsd:complexType>
                    <xsd:sequence>
```

```
<xsd:element name="color" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:attribute name="name" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="R"/>
          <xsd:enumeration value="W"/>
          <xsd:enumeration value="B"/>
          <xsd:enumeration value="Y"/>
          <xsd:enumeration value="Dark"/>
          <xsd:enumeration value="Other"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="price">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="factory_price" type="xsd:string"/>
      <xsd:element name="retail_price" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="wheels" type="xsd:int" />
</xsd:sequence>
<xsd:attribute name="name" use="required">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="BEETLE"/>
      <xsd:enumeration value="SEDAN"/>
      <xsd:enumeration value="JEEP"/>
      <xsd:enumeration value="TOPOLINO"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
<xsd:attribute name="company" type="xsd:IDREF" use="required">
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

XSD – Separate Complex Types Style

```

<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified" elementFormDefault="qualified" version="1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="oldcarsworld" type="oldcarsworldType" />
  <xsd:complexType name="oldcarsworldType">
    <xsd:sequence>
      <xsd:element name="company" minOccurs="0" maxOccurs="unbounded" type="companyType" />
      <xsd:element name="car" maxOccurs="unbounded" type="carType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="carType">
    <xsd:sequence>
      <xsd:element name="model" type="xsd:string" minOccurs="0"/>
      <xsd:element name="horsepower" type="horsepowerType" />
      <xsd:element maxOccurs="unbounded" name="production" type="productionType" minOccurs="0"/>
      <xsd:element name="color" type="colorType" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element name="price" type="priceType" />
      <xsd:element name="wheels" type="xsd:int" />
    </xsd:sequence>
    <xsd:attribute name="name" type="car names Type" use="required"/>
    <xsd:attribute name="company" type="xsd:IDREF" use="required"/>
  </xsd:complexType>
  <xsd:complexType name="priceType">
    <xsd:sequence>
      <xsd:element name="factory_price" type="xsd:string" />
      <xsd:element name="retail_price" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="car names Type">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="BETTER" />
        <xsd:enumeration value="BETTER" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:complexType>
  <xsd:complexType name="horsepowerType">
    <xsd:choice>
      <xsd:element name="HP" type="xsd:int"/>
      <xsd:sequence>
        <xsd:element name="HPmin" type="xsd:int" minOccurs="0"/>
        <xsd:element name="HPmax" type="xsd:int" minOccurs="0"/>
      </xsd:sequence>
    </xsd:choice>
  </xsd:complexType>
  <xsd:complexType name="companyType">
    <xsd:sequence>
      <xsd:element name="country" type="xsd:string" minOccurs="0"/>
      <xsd:attribute name="name" type="xsd:ID" use="required"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="productionType">
    <xsd:sequence>
      <xsd:element name="start" type="xsd:int" minOccurs="0"/>
      <xsd:element name="finish" type="xsd:int" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="colorType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="R"/>
      <xsd:enumeration value="W"/>
      <xsd:enumeration value="B"/>
      <xsd:enumeration value="Y"/>
      <xsd:enumeration value="Dark"/>
      <xsd:enumeration value="Other"/>
    </xsd:restriction>
  </xsd:complexType>
</xsd:schema>

```

XML, DTD, Schema validations

- XML Syntax Validation: <http://www.utilities-online.info>
- DTD Syntax Validation: <http://www.validome.org/grammar/>
- Schema Syntax Validation: <http://www.utilities-online.info>

- XML against DTD Validation: http://www.w3schools.com/xml/xml_validator.asp
- XML against Schema Validation: <http://www.utilities-online.info>
- XML, DTD, Schema Validation: <http://www.xmlvalidation.com>

- Making DTD out of XML : http://www.w3schools.com/xml/xml_validator.asp
- Making Schema out of XML :
http://www.xmlforasp.net/codebank/system_xml_schema/buildschema/buildxmlschema.aspx

XPath outlines

- XPath is a language for finding information in an XML document.
- XPath uses path expressions to select **nodes** or **node-sets** in an XML document. The node is selected by following a path or steps.

	selects	example	explanation
node	all children of <i>node</i>	collection	all children of root element collection
/	from the root node	/collection collection/car /collection/car[1] /collection/car[last()] /collection/car[position()<3] /collection/car[price<1000] /collection/car[price<1000]/top-speed	root element collection all car elements that are children of collection first car which is child of collection last car which is child of collection first two cars which are child of collection all cars with a price element with a value below 1000 top speed of all the cars with prices below 1000
//	all elements from the current node	//car collection//car //car[@producer] //car[@producer='Ford']	all the cars in the XML file all cars descendant of collection all cars with an attribute <i>producer</i> all cars produced by Ford
.	current node	//car//top-speed[@units='\"km/h\" and . > 100]/../name	name of the cars with top-speed above 100 km/h
..	parent of current node	//producer[@country='\"Italy\"']/../name	name of the all the cars produced in Italy
@	attribute	//@producer	all attributes with name producer
 	several paths	//car/name //car/price //name //price	all name and price elements of all cars all name and price elements in the XML file

Online testing Xpath expressions: <http://www.freeformatter.com/xpath-tester.html#ad-output>

- Use of XPath: See some examples at <http://www.zvon.org/xxl/XPathTutorial/General/examples.html>
`xml.evaluate(XPath, xmlDoc, null, XPathResult.ANY_TYPE, null);`

XPath use in HTML Script

1. XML file

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<collection>

<car category="Classic" source="http://www.topclassiccars.com/austin-7.html">
  <name>Austin 7</name>
  <nickname>Baby Austin</nickname>
  <production>
    <producer country="United Kingdom">Austin Motor Company</producer>
    <start>1922</start>
    <end>1939</end>
  </production>
  <production>
    <producer country="Germany">BMW</producer>
    <start>1927</start>
  </production>
  <top-speed units="km/h">77</top-speed>
  <displacement>747 cc</displacement>
  <price date="1930" currency="$">445.00</price>
</car>

<car category="Classic" source="http://www.topclassiccars.com/pontiac-bonneville.html">
  <name>Pontiac Bonneville</name>
  <nickname>Bonneville</nickname>
  <production>
    <producer country="USA">GM</producer>
    <start>1959</start>
    <end>1970</end>
  </production>
  <top-speed units="mph">120</top-speed>
  <top-speed units="km/h">192</top-speed>
  <displacement>6,374 cc</displacement>
</car>

<car category="Modern" source="http://www.topclassiccars.com/maserati-bora.html">
  <name>Maserati Bora</name>
  <production>
    <producer country="Italy">Maserati</producer>
    <start>1971</start>
    <end>1978</end>
  </production>
  <top-speed units="km/h">285</top-speed>
  <displacement>4,719 cc</displacement>
  <price date="2009" currency="$">30,000</price>
</car>

<car category="Classic" source="http://www.topclassiccars.com/ferrari-512-bb-le-mans.html">
  <name>Ferrari 512 BB</name>
  <nickname>Ferrari 512 Le Mans</nickname>
  <production>
    <producer country="Italy">Ferrari</producer>
    <start>1975</start>
    <end>1980</end>
  </production>
  <top-speed units="mph">203</top-speed>
  <top-speed units="km/h">325</top-speed>
  <displacement>4,942 cc</displacement>
</car>

</collection>
```

See some examples at <http://zvon.org/xxl/XPathTutor>

2. HTML (source)

```
<html>
<body>
<script type="text/javascript">
function loadXMLDoc(dname)
{
if (window.XMLHttpRequest)
{
  xhttp=new XMLHttpRequest();
}
else
{
  xhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
xhttp.open("GET",dname,false);
xhttp.send("");
return xhttp.responseXML;
}

query=1;

xml=loadXMLDoc("car-collection.xml");
path='collection/car/name';

var nodes=xml.evaluate(path, xml, null, XPathResult.ANY_TYPE, null);
var result=nodes.iterateNext();

document.write("<b>Query "+(query++)+": "+path+"</b><br />");

while (result)
{
  document.write(result.firstChild.nodeValue);
  document.write("<br />");
  result=nodes.iterateNext();
}

</script>
</body>
</html>
```

Query 1: collection/car/name

Austin 7
Pontiac Bonneville
Maserati Bora
Ferrari 512 BB

3. HTML (view)

Query 2: //car/top-speed

77
120
192
285
203
325

Query 3: //car/producer

Austin Motor Company
BMW
GM
Maserati
Ferrari

Query 4: /collection/car[2]/name

Pontiac Bonneville

Query 5: /collection/car[last()-2]/name

Pontiac Bonneville

Query 6: //car[price/@currency='\$' and price<1000]/name

Austin 7

Query 7: //car[@category='Classic' and price<1000]/price/@date

1930

Query 8: //car/price/text()

XPath additional functions

<code>number(<i>arg</i>)</code>	numeric value of <i>arg</i>
<code>abs(<i>num</i>)</code>	absolute value of <i>num</i>
<code>ceiling(<i>num</i>)</code>	smallest integer that is greater than <i>num</i>
<code>floor(<i>num</i>)</code>	largest integer that is not greater than <i>num</i>
<code>round(<i>num</i>)</code>	the nearest integer to <i>num</i>
<code>string(<i>arg</i>)</code>	string value of <i>arg</i>
<code>compare(<i>c1</i>,<i>c2</i>)</code>	-1 if <i>c1</i> less than <i>c2</i> , 0 if <i>c1</i> equal to <i>c2</i> , or 1 if <i>c1</i> greater than <i>c2</i>
<code>concat(<i>str1</i>,...)</code>	concatenation of strings
<code>string-join((<i>str1</i>,<i>str2</i>,...),<i>sep</i>)</code>	concatenation of strings using <i>sep</i> as separator
<code>substring(<i>str</i>,<i>start</i>[,<i>len</i>])</code>	substring from <i>start</i> position to the [<i>end</i> or specified length <i>len</i>]
<code>string-length(<i>str</i>)</code>	length of <i>str</i>
<code>upper-case(<i>string</i>)</code>	string argument to upper-case (lower-case)
<code>contains(<i>s1</i>,<i>s2</i>)</code>	true if string <i>s1</i> contains <i>s2</i> , false otherwise
<code>starts-with(<i>s1</i>,<i>s2</i>)</code>	true if <i>s1</i> starts with <i>s2</i> , otherwise false (ends-with)
<code>count(<i>item</i>,<i>item</i>,...)</code>	count of nodes
<code>avg(<i>arg</i>,<i>arg</i>,...)</code>	average of the argument numeric values
<code>max(<i>arg</i>,<i>arg</i>,...)</code>	argument that is greater than the others (min)
<code>sum(<i>arg</i>,<i>arg</i>,...)</code>	sum of the numeric value of each node
<code>id(<i>s1</i>, <i>s2</i>, ...)</code>	nodes whose ID matches some of the strings <i>s1</i> , <i>s2</i> , ...
<code>idref(<i>s1</i>, <i>s2</i>, ...)</code>	sequence of element or attribute nodes with IDREF value one of the values string
<code>position()</code>	index position of the current node
<code>last()</code>	last element in the current node

XSL Outlines

- XSL (eXtensible Stylesheet Language) is a style sheet language for XML documents.
- XSL is used to transform XML documents into other formats, like XHTML

<xsl:template match=“*xPath expression*”>

“defines a template matched to an XML element resulting from a xpath query to an XML file”

<xsl:value-of select=“*xPath expression*”>

“extracts the value of an XML element resulting from a xpath query to an XML element”

<xsl:for-each select=“*xPath expression*”>

“selects every XML element of a specified node-set resulting from a xpath query”

<xsl:sort select=“*xPath expression*”/>

“introduced in a <xsl:for-each> it provides an order according to the xml element obtained from the XPath expression”

<xsl:if test=“*condition*”>

“puts a conditional test against the content of the XML file”

<xsl:choose>

“contains <xsl:when test=“condition”> and <xsl:otherwise> to express multiple conditional tests”

XSL use in HTML script

1. XML file: data

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Edited by XMLSpy® -->
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
  <cd>
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
    <country>UK</country>
    <company>CBS Records</company>
    <price>9.90</price>
    <year>1988</year>
  </cd>
  <cd>
    <title>Greatest Hits</title>
    <artist>Dolly Parton</artist>
    <country>USA</country>
    <company>A&M Records</company>
    <price>12.90</price>
    <year>1980</year>
  </cd>
</catalog>
```

2. XSL file: view

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Edited by XMLSpy® -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
  <body>
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th>Title</th>
        <th>Artist</th>
      </tr>
      <xsl:for-each select="catalog/cd">
        <tr>
          <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="artist"/></td>
        </tr>
      </xsl:for-each>
    </table>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

3. HTML file (source)

```
<html>
<head>
<script>
function loadXMLDoc(dname)
{
  if (window.XMLHttpRequest)
  {
    xhttp=new XMLHttpRequest();
  }
  else
  {
    xhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
  xhttp.open("GET",dname,false);
  xhttp.send("");
  return xhttp.responseXML;
}

function displayResult()
{
  xml=loadXMLDoc("cdcatalog.xml");
  xsl=loadXMLDoc("cdcatalog.xsl");
  // code for IE
  if (window.ActiveXObject)
  {
    ex=xsl.transformNode(xml);
    document.getElementById("example").innerHTML=ex;
  }
  // code for Mozilla, Firefox, Opera, etc.
  else if (document.implementation && document.implementation.createDocument)
  {
    xsltProcessor=new XSLTProcessor();
    xsltProcessor.importStylesheet(xsl);
    resultDocument = xsltProcessor.transformToFragment(xml,document);
    document.getElementById("example").appendChild(resultDocument);
  }
}
</script>
</head>
<body onload="displayResult()">
<div id="example" />
</body>
</html>
```

My CD Collection

Title	Artist
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler
Greatest Hits	Dolly Parton
Still got the blues	Gary Moore
Eros	Eros Ramazzotti
One night only	Bee Gees
Sylvias Mother	Dr.Hook
Maggie May	Rod Stewart
Romanza	Andrea Bocelli
When a man loves a woman	Percy Sledge
Black angel	Savage Rose
1999 Grammy Nominees	Many
For the good times	Kenny Rogers
Big Willie style	Will Smith
Tupelo Honey	Van Morrison
Soulsville	Jorn Hoel
The very best of	Cat Stevens
Stop	Sam Brown
Bridge of Spies	T' Pau
Private Dancer	Tina Turner
Midt om natten	Kim Larsen
Pavarotti Gala Concert	Luciano Pavarotti
The dock of the bay	Otis Redding
Picture book	Simply Red
Red	The Communards
Unchain my heart	Joe Cocker

4. HTML file (view)

RDF: Resource Description Framework

- RDF was designed for **describing resources on the web**.
- RDF is to be read and understood by computers.
- RDF is not for being displayed to people.
- RDF is written in XML.

- RDF uses Uniform Resource Identifiers (URIs).
- RDF basic concepts: $P(R)=V$
 - **Resources**: anything that can have a URI.
 - **Properties**: resource that has a name.
 - **Property values**: the value of a Property for a resource. It can be another resource.
- RDF **statements**: $P(S)=O$
 - Subject (S): the resource of the statement.
 - Predicate (P): the property of the statement.
 - Object (O): the property value of the statement.
- Example: “The webmaster of <http://invented.page> is John Smith”
Webmaster(<http://invented.page>)=John Smith



RDF Outlines

- `<rdf:RDF [xmlns:X1="URI1" ...]>` is the root element of an RDF document (unique) and, optionally, it indicates the name spaces X_i used inside the `rdf:RDF`.
- **`<rdf:Description>` is the statement constructor that identifies a resource (subject) with the about attribute and contains elements (predicates) that describe the resource.**
 - `<rdf:Bag>` describes a list of values that is intended to be unordered.
 - `<rdf:Seq>` describes a list of values that is intended to be ordered.
 - `<rdf:Alt>` describes a list of alternative values.

Ex. `<rdf:Description rdf:about="http://www.old_cars.org/Beetle">`

`<producer> Volkswagen</producer>`

`<start-prod>1938</start-prod>`

...

`</rdf:Description>`

`<rdf:Description rdf:about="http://www.old_cars.org">`

`<car>`

`<rdf:Alt>`

`<rdf:li>Beetle</rdf:li>`

`<rdf:li>Sedan</rdf:li>`

`<rdf:li>Jeep</rdf:li>`

...

`<rdf:Alt>`

`</car>`

`<owner rdf:resource="http://www.old_cars.org/People/">`

`</rdf:Description>`

`producer (www.old_cars.org/Beetle) = "Volkswagen"`
`start-prod(www.old_cars.org/Beetle) = "1938"`
 ...

`car (www.old_cars.org) = ("Beetle" | "Sedan" | "Jeep" | ...)`
`owner (www.old_cars.org) = http://www.old_cars.org/People/`

See some examples at <http://zvon.org/xxl/RDFTutorial/General/book.html>

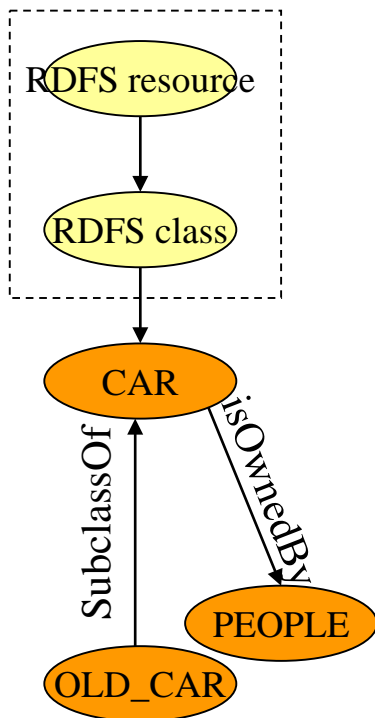


RDF Example

Number		Object
1	<code><?xml version="1.0"?></code>	
2	<code><rdf:RDF</code>	
3	<code>xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:cd="http://www.recshop.fake/cd#"></code>	
4	<code><rdf:Description</code>	
5	<code>rdf:about="http://www.recshop.fake/cd/Empire_Burlesque" cd:artist="Bob Dylan"></code>	"Bob Dylan"
6	<code><!-- property as attribute --></code>	"USA"
7	<code><cd:country>USA</cd:country></code>	http://www.recshop.fake/cd/columbia
8	<code><!-- property as element --></code>	
9	<code><cd:company</code>	http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag
10	<code>rdf:resource="http://www.recshop.fake/cd/columbia"/></code>	genid:A116295
11	<code><!-- property as resource --></code>	
12	<code></rdf:Description></code>	"John"
13	<code><rdf:Description</code>	
14	<code>rdf:about="http://www.recshop.fake/cd/Beatles"></code>	"Paul"
15	<code><cd:artist></code>	
16	<code><!-- artist is an unordered list of names --></code>	"George"
17	<code><rdf:Bag></code>	
18	<code><rdf:li>John</rdf:li></code>	
19	<code><rdf:li>Paul</rdf:li></code>	"Ringo"
20	<code><rdf:li>George</rdf:li></code>	
21	<code><rdf:li>Ringo</rdf:li></code>	
22	<code></rdf:Bag></code>	
23	<code></cd:artist></code>	http://www.w3.org/1999/02/22-rdf-syntax-ns#Alt
24	<code><cd:format></code>	genid:A116296
25	<code><!-- format is an alternative list of cd/record/tape --></code>	
26	<code><rdf:Alt></code>	"CD"
27	<code><rdf:li>CD</rdf:li></code>	
28	<code><rdf:li>Record</rdf:li></code>	
29	<code><rdf:li>Tape</rdf:li></code>	"Record"
30	<code></rdf:Alt></code>	
31	<code></cd:format></code>	
32	<code></rdf:Description></code>	"Tape"
33	<code></rdf:RDF></code>	

RDF Schema (RDFS)

- RDFS is **an extension to RDF** to define classes/subclasses/instances.
- RDFS provides the framework to describe application-specific classes and properties.
- Classes in RDFS allows resources to be defined as instances or subclasses of classes.



Ex.

```

<?xml version="1.0"?>

<rdf:RDF
    xmlns:rdf= "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    (always the same)
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    (the same)
    xml:base= "http://www.old_cars.org/cars#"
    (db)
    <!-- indicates that this is an RDF document -->
    <!-- namespace of RDF -->
    <!-- namespace of RDFS (always the same) -->
    <!-- namespace of base (URI of cars db) -->

    <rdf:Description rdf:ID="car">
        <rdf:type
            rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
        <!-- car is a class (type = instance of Class) -->
        <!-- alternatively: <rdfs:Class rdf:ID="car"/> -->
    </rdf:Description>

    <rdf:Description rdf:ID="old_car">
        <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
        <rdfs:subClassOf rdf:resource="#car"/>
        <!-- old_car is a subclass of car and an instance of Class -->
    </rdf:Description>

    ...

    <rdf:Description rdf:ID="is-owned-by">
        <rdf:type rdf:resource="rdf:Property"/>
        <rdfs:domain rdf:resource="#car"/>
        <rdfs:range rdf:resource="#people"/>
        <!-- property saying cars are-owned-by people -->
    </rdf:Description>

    ...
</rdf:RDF>
  
```

rdfs:subClassOf defines the class hierarchy
rdf:type defines the instances of a class

W3C provides a RDF validator at <http://www.w3.org/RDF/Validator/>



RDFS constructors

PROPERTIES

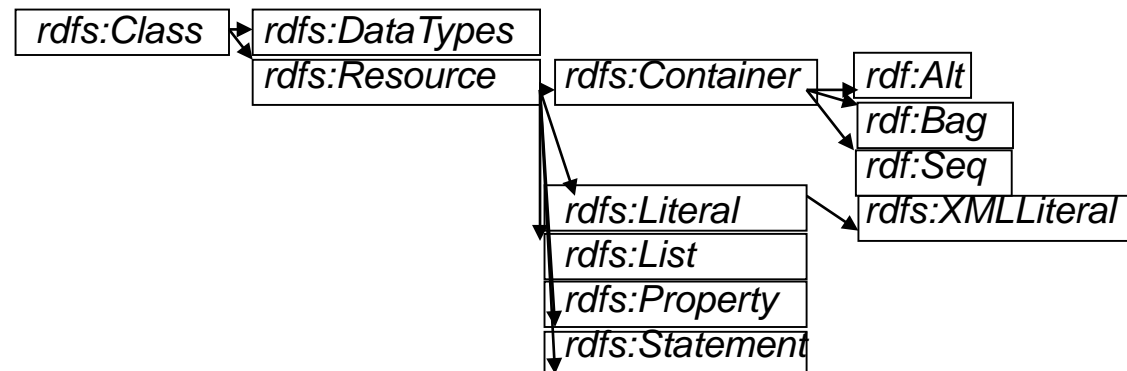
- `rdfs:domain`
- `rdfs:range`
- `rdfs:subPropertyOf`
- `rdfs:subClassOf`
- `rdfs:comment`
- `rdfs:label`
- `rdfs:isDefinedBy`
- `rdfs:seeAlso`

CLASSES

- `rdfs:Class`
- `rdfs:DataType`
- `rdfs:Literal`
- `rdfs:Container`
- `rdfs:ConstraintResource`
- `rdfs:ConstraintProperty`

Element	Domain	Range	Description
<code>rdfs:subClassOf</code>	Class	Class	The resource is a subclass of a class
<code>rdfs:domain</code> <code>rdfs:range</code>	Property	Class	The domain of the resource The range of the resource
<code>rdfs:subPropertyOf</code>	Property	Property	The property is a sub property of a property
<code>rdfs:comment</code> <code>rdfs:label</code>	Resource	Literal	Human readable description of the resource Human readable label (name) of the resource
<code>rdfs:isDefinedBy</code> <code>rdfs:seeAlso</code> <code>rdfs:member</code>	Resource	Resource	The definition of the resource The additional information about the resource The member of the resource

Hierarchy of rdfs Classes:



SPARQL

- SPARQL Protocol and RDF Query Language
- Emulates SQL
- To write queries on data that follow the RDF specification.
 - (subject, predicate, object): equivalent to (entity id, att name, att value) with entity id and att name as URI's. Example: (<http://www.company.com/hr/emp3>, <http://www.w3.org/2006/vcard/ns#title>, "Vice President")
 - Prefixing: long URI prefixes can be named for simplification:
 - @prefix vcard: <<http://www.w3c.org/2006/vcard/ns#>>.
 - @prefix sn: <<http://www.company.co/hr/>>.
 - (sn:emp3 vcard:title "Vice President")
- Queries 1: recovering data (all data satisfying all the where conditions)


```
SELECT list-of-variables
WHERE {
    ?variable att-name att-value-or-variable .
}
```
- Queries 2: filters


```
SELECT list-of-variables
WHERE {
    ?variable att-name att-value-or-variable .
    FILTER (condition-on-variables)
}
```
- Queries 3: optional fields (all data satisfying all the where conditions and optionally the optional conditions)


```
SELECT list-of-variables
WHERE {
    ?variable att-name att-value-or-variable .
    OPTIONAL (?variable att-name att-value-or-variable . )
}
```
- Query 4: not existing (all data satisfying all the where conditions and not having the not exists conditions)


```
SELECT list-of-variables
WHERE {
    ?variable att-name att-value-or-variable .
    NOT EXISTS (?variable att-name att-value-or-variable . )
}
```
- Query 5: bind to assign a value to a variable (ex., value concat(?first-name, " ", ?second-name) and display this value in results).


```
SELECT list-of-variables
WHERE {
    ?variable att-name att-value-or-variable .
    BIND (value-or-variable AS ?variable )
}
```

OWL: The Web Ontology Language



- OWL is built on top of RDF and written in XML.
- OWL is for processing information on the web.
- OWL was designed to be interpreted by computers and not for being read by people
- OWL has three sublanguages
 - **OWL full** OWL syntax + RDF (complete expressiveness without computational guarantees)
 - **OWL DL** restricted to FOL fragment (computational complete & decidable reasoning K)
 - **OWL Lite** is “easier to implement” subset of OWL DL (hierarchical K)
- Semantic layering
 - OWL DL (Description Logic) \approx OWL full **within DL fragment**
 - DL semantics **officially definitive**
- OWL DL based on **SHIQ** Description Logic
 - In fact it is equivalent to **SHOIN(D_n)** DL
- OWL DL Benefits from many years of DL research
 - Well defined **semantics**
 - **Formal properties** well understood (complexity, decidability)
 - Known **reasoning algorithms**
 - **Implemented systems** (highly optimised)

OWL: Class Constructors and Axioms

Constructor	DL Syntax	Example	Modal Syntax	
intersectionOf	$C_1 \sqcap \dots \sqcap C_n$	Human \sqcap Male	$C_1 \wedge \dots \wedge C_n$	Men
unionOf	$C_1 \sqcup \dots \sqcup C_n$	Doctor \sqcup Lawyer	$C_1 \vee \dots \vee C_n$	things that are either Drs. or Layers, or both
complementOf	$\neg C$	\neg Male	$\neg C$	things that are not males
oneOf	$\{x_1\} \sqcup \dots \sqcup \{x_n\}$	$\{\text{john}\} \sqcup \{\text{mary}\}$	$x_1 \vee \dots \vee x_n$	John or Mary
allValuesFrom	$\forall P.C$	$\forall \text{hasChild}.\text{Doctor}$	$[P]C$	things that all their children are doctors
someValuesFrom	$\exists P.C$	$\exists \text{hasChild}.\text{Lawyer}$	$\langle P \rangle C$	things that some of their children is a layer
maxCardinality	$\leq nP$	$\leq 1 \text{hasChild}$	$[P]_{n+1}$	things with one or less children
minCardinality	$\geq nP$	$\geq 2 \text{hasChild}$	$\langle P \rangle_n$	things with two or more children

Axiom	DL Syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human \sqsubseteq Animal \sqcap Biped
equivalentClass	$C_1 \equiv C_2$	Man \equiv Human \sqcap Male
disjointWith	$C_1 \sqsubseteq \neg C_2$	Male $\sqsubseteq \neg$ Female
sameIndividualAs	$\{x_1\} \equiv \{x_2\}$	$\{\text{President_Bush}\} \equiv \{\text{G_W_Bush}\}$
differentFrom	$\{x_1\} \sqsubseteq \neg \{x_2\}$	$\{\text{john}\} \sqsubseteq \neg \{\text{peter}\}$
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter \sqsubseteq hasChild
equivalentProperty	$P_1 \equiv P_2$	cost \equiv price
inverseOf	$P_1 \equiv P_2^-$	hasChild \equiv hasParent ⁻
transitiveProperty	$P^+ \sqsubseteq P$	ancestor ⁺ \sqsubseteq ancestor
functionalProperty	$\top \sqsubseteq \leq 1P$	$\top \sqsubseteq \leq 1 \text{hasMother}$
inverseFunctionalProperty	$\top \sqsubseteq \leq 1P^-$	$\top \sqsubseteq \leq 1 \text{hasSSN}^-$

OWL: Example

Person $\sqcap \forall \text{hasChild}. (\text{Doctor} \sqcup \exists \text{hasChild}.\text{Doctor})$

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="collection">
    <owl:Class rdf:about="#Person"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasChild"/>
      <owl:toClass>
        <owl:unionOf rdf:parseType="collection">
          <owl:Class rdf:about="#Doctor"/>
          <owl:Restriction>
            <owl:onProperty rdf:resource="#hasChild"/>
            <owl:hasClass rdf:resource="#Doctor"/>
          </owl:Restriction>
        </owl:unionOf>
      </owl:toClass>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

persons with all their children or doctors
or fathers of some doctor



- OWL 2 is designed to facilitate ontology development and sharing via the Web, with the ultimate goal of making Web content more accessible to machines
- Extensions:
 - keys;
 - property chains;
 - richer datatypes, data ranges;
 - qualified cardinality restrictions;
 - asymmetric, reflexive, and disjoint properties; and
 - enhanced annotation capabilities
- **DisjointUnion** defines a class as the union of other classes, all of which are pairwise disjoint.
- **DisjointClasses** states that all classes from the set are pairwise disjoint.
- **NegativeObjectPropertyAssertion** (resp. **NegativeDataPropertyAssertion**) states that a given property does not hold for the given individuals (resp. literal).
- A class expression defined using an **ObjectHasSelf** restriction denotes the class of all objects that are related to themselves via the given object property.
- **ObjectMinCardinality**, **ObjectMaxCardinality**, and **ObjectExactCardinality** (respectively, **DataMinCardinality**, **DataMaxCardinality**, and **DataExactCardinality**) allow for the assertion of minimum, maximum or exact qualified cardinality restrictions on object (respectively, data) properties.
- **ReflexiveObjectProperty** allows us to assert that an object property expression is globally reflexive - that is, the property holds for all individuals. **IrreflexiveObjectProperty** allows us to assert that an object property expression is irreflexive - that is, the property does not hold for any individual. **AsymmetricObjectProperty** allows us to assert that an object property expression is asymmetric - that is if the property expression OPE holds between the individuals x and y, then it cannot hold between y and x.
- **DisjointObjectProperties** allows us to assert that several object properties are pairwise incompatible (exclusive); that is, two individuals cannot be connected by two different properties of the set.
- **ObjectPropertyChain** states that any individual x connected with an individual y by a chain of object properties expressions OPE1, ..., OPE_n is necessary connected with y by the object property OPE.
- **HasKey** axiom states that each *named* instance of a class is uniquely identified by a (data or object) property or a set of properties - that is, if two named instances of the class coincide on values for each of key properties, then these two individuals are the same.
- OWL 2 *datatypes* include a) various kinds of numbers, adding support for a wider range of XML Schema Datatypes (double, float, decimal, positiveInteger, etc.) and providing its own datatypes, e.g., owl:real; b) strings with (or without) a Language Tag (using the rdf:PlainLiteral datatype); and c) boolean values, binary data, IRIs, time instants, etc.
- **DatatypeRestriction** also makes it possible to specify restrictions on datatypes by means of constraining *facets* that constrain the range of values allowed for a given datatype, by length (for strings) e.g., minLength, maxLength, and minimum/maximum value, e.g., minInclusive, maxInclusive.

References

1. Davenport, T.H.; Prusak, L. (2000): *Working knowledge. How organizations manage what they know*, 2nd Edition.
2. Brachman, R.J.; Levesque, H.J. (2003): *Knowledge Representation and Reasoning*.
3. UML Diagrams: <http://www.uml-diagrams.org/uml-25-diagrams.html>
4. Miksch, M.L. (1974): *A framework for representing knowledge*,.
5. Schank, R.C.; Abelson, R.P. (1977): *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures*.
6. Sowa, J.F. (1992): *Semantic Networks*.
7. Noy, N.; Rector, A. (2006): *Defining N-ary Relations on the Semantic Web*.
8. Noy, N; McGuinness, D. (2001): *Ontology Development 101: A Guide to Creating Your First Ontology*.

END

In this course you should have learnt:

1. What is knowledge?
2. What sorts of knowledge do exist?
3. What is knowledge representation?
4. What is knowledge engineering?
5. How to formalize knowledge with First Order Logic?
6. How to formalize knowledge with Rules?
7. How to formalize knowledge with Production Rules?
8. How to formalize knowledge with Frames and Scripts?
9. How to formalize knowledge with Semantic Networks?
10. How to formalize knowledge with Ontologies?
11. How to use Protégé to construct Ontologies?
12. What is a Knowledge LifeCycle?
13. Which processes compose a Knowledge LifeCycle?
14. How to Audit Knowledge?
15. How to Acquire Knowledge?
16. How to Deploy Knowledge Systems?
17. How to produce a Knowledge-Base (ontology) methodologically?