Artificial Intelligence 3

Lecture Notes

for

Knowledge Representation and Inference I

Don Sannella

Department of Artificial Intelligence

University of Edinburgh

The knowledge representation problem

Introduction

The Knowledge Representation problem, stated informally, is: How can we represent knowledge in such a way that a machine can do something useful with it?

Before answering this question, we have to consider the question: What is knowledge? There are (at least) four kinds of knowledge we will be interested in:

- *Specific knowledge*, i.e. knowledge about specific things (in the form of *propositions*). For example:
  My car is red.
  Janet has a book.
- *General knowledge*, i.e. generalizations. For example:
  Every human has a mother.
  All Italians eat spaghetti.
- *Procedural knowledge*, i.e. knowledge about "how to". For example:
  To make a cup of tea, first find a cup and some tea. Then, ...
- *Meta-knowledge*, i.e. knowledge about knowledge. For example:
  Travel agents know about airline timetables.

"Knowledge" would be better referred to as *belief*. It is possible to "know" something which is false. Refining this: "knowledge" in our sense is usually *belief supported by evidence of some kind* (e.g. the chain of reasoning from other facts which led to it).

The use of knowledge in "intelligent" programs

Almost any AI program has knowledge of some kind -- some people would go so far as to *define* artificial intelligence like this.

Expert systems embody knowledge about some specialized domain. For example, the MYCIN system (1976) is intended to assist with the diagnosis and treatment of blood infections. MYCIN assists a physician who is not a specialist in antibiotics and blood infections to select a therapy based on what it can infer from tests, etc.

The knowledge representation problem

Ingredients of a knowledge base:

- A *representation language* for expressing the knowledge
- An *inference regime* for deducing things which are implicit in the knowledge -- generally, there are infinitely many useful implicit consequences of the explicit knowledge
- Some *knowledge*

Simultaneously producing good candidates for all three of these is the central problem.

Desirable properties:

- *Expressive adequacy:* it should be possible to express everything we need to express.
- *Reasoning efficiency:* inference should not be intolerably slow.

These are mutually antagonistic goals, unfortunately, so this is a difficult tradeoff.

History

Aristotle mentioned logic. But Leibniz (1646-1716) was the first to talk about a calculus for representing and manipulating ideas in an unambiguous way just as numbers can be represented and manipulated.

Turn of the century: Frege, Russell, Peano invented symbolic logic.

More recently, philosophers and others have discussed different kinds of logic for expressing different kinds of information.

First-order logic

First-order logic is one of the most popular knowledge representation formalisms. Here are some examples of specific and general knowledge encoded in first-order logic:

My car is red.
red (GSX638T)
Janet has a book.
$\exists b.(book(b) \wedge has(Janet,b))$
Every human has a mother.
$\forall h.(human(h) \Rightarrow \exists m.mother(m,h))$
All Italians eat spaghetti.
$\forall h.(Italian(h) \Rightarrow eat(h,spaghetti))$

Inference rules can be used to infer new facts from known facts. Here are two examples:

$$\frac{\forall x.P(x)}{P(a)}$$

$$\frac{P \Rightarrow Q \qquad P}{Q}$$

So if we know "All Italians eat spaghetti" and "Luigi is an Italian" we can infer "Luigi eats spaghetti":

$$\frac{\dfrac{\forall h.(\text{Italian}(h) \Rightarrow \text{eat}(h,\text{spaghetti}))}{\text{Italian}(\text{Luigi}) \Rightarrow \text{eat}(\text{Luigi},\text{spaghetti})} \qquad \text{Italian}(\text{Luigi})}{\text{eat}(\text{Luigi},\text{spaghetti})}$$

Some problems

Meta-knowledge:

The problem is that predicate calculus is for talking about *things* (people, objects) and a fact is not a thing. For example, consider the problem of expressing "Pat knows the combination of that safe". Suppose the combination is 42-25-17; knows(Pat,42-25-17) is not good enough, because if the combination of some safe in Abu Dhabi is also 42-25-17 we cannot infer that Pat knows its combination as well.

Almost-universal facts:

Birds can fly (except penguins), all humans have mothers (except Adam and Eve), if I throw something in the air it will fall back to the ground (unless it is hit by lightning, is caught by a bird, etc.). The fact that almost nothing is really universally true causes problems because we don't want to qualify every almost-universal fact with all its rare exceptions.

Procedural knowledge

Events which change the world:

These are particularly hard to represent when the change is continuous (e.g tree growing) rather than instantaneous.

Probabilistic knowledge (like in MYCIN)

Summary

It is important for "intelligent" programs (expert systems, language understanding systems, etc.) to be able to express knowledge about the world in some explicit symbolic way so that inferences can be drawn from it without too much work.

There is a range of problems which make this difficult because the kinds of things to be represented are so varied.

There are several competing methods, each of which is good for certain kinds of knowledge and certain kinds of inference. We will examine three main approaches in this course.

There is no known universally "best" method -- this is probably the central problem of AI.

Propositional logic

References:

Lemmon, *Beginning Logic*

When somebody uses an argument to prove something, he is normally asserting the premises to be true and also stating that the conclusion is true *on the strength of* the premises.

It is possible to infer true things from false premises using a sound argument:

Napolean was German. All Germans are European.
∴ Napolean was European.

It is possible to infer false things from false premises using a sound argument:

Napolean was German. All Germans are Asiatic.
∴ Napolean was Asiatic.

It is possible to infer false things from true premises, but only using an *unsound* argument:

Napolean was French. All Frenchmen are Europeans.
∴ The moon is made of green cheese.

It is also possible to infer true things from true premises using an unsound argument, but just because the premises and conclusion are true doesn't mean that the argument is sound:

Napolean was French. All Frenchmen are Europeans.
∴ Burns was Scottish.

The point of logic is to distinguish sound arguments from unsound ones.

Propositional logic

Propositional logic views statements like "Jack has 3 books" and "AI is wonderful" as atomic. More complex statements can be built up from these using conjunction, implication, etc.

A well-formed formula (wff) is either:

- an atomic proposition P, or
- an expression of the form
  $P \wedge Q$, $P \vee Q$, $\neg P$, or $P \Rightarrow Q$
  where P, Q are wffs.

Translating sentences to wffs

If it is not raining, then either I am dry or I am swimming.

$\neg R \Rightarrow (D \vee S)$

where R = it is raining, D = I am dry, S = I am swimming.

If it is raining, then it is not the case that if it is not snowing it is not raining.

$R \Rightarrow \neg (\neg S \Rightarrow \neg R)$

where R = it is raining, S = it is snowing

See Lemmon for many more examples.

Truth tables

These can be used to check if a formula is a tautology (true regardless of the truth of the atomic propositions which occur in it).

The connectives $\neg, \wedge, \vee, \Rightarrow$ are defined by the following truth tables:

P	$\neg P$
T	F
F	T

P	Q	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$
T	T	T	T	T
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Truth tables for complex formulae can be built by reference to these.

P	Q	R	$\neg Q$	$P \wedge \neg Q$	$(P \wedge \neg Q) \vee R$	$R \wedge \neg Q$	$(R \wedge \neg Q) \Rightarrow P$	$((P \wedge \neg Q) \vee R) \Rightarrow ((R \wedge \neg Q) \Rightarrow P)$
T	T	T	F	F	T	F	T	T
T	T	F	F	F	F	F	T	T
T	F	T	T	T	T	T	T	T
T	F	F	T	T	T	F	T	T
F	T	T	F	F	T	F	T	T
F	T	F	F	F	F	F	T	T
F	F	T	T	F	T	T	F	F
F	F	F	T	F	T	F	T	T

So $((P \wedge \neg Q) \vee R) \Rightarrow ((R \wedge \neg Q) \Rightarrow P)$ is <u>not</u> a tautology (although $((P \wedge \neg Q) \vee R) \Rightarrow ((R \wedge \neg Q) \Rightarrow \neg P)$ is).

See Lemmon for more examples.

Proofs in propositional logic

The inference rules are listed below. Let A, B, C be wffs and let L, L', L'' be sets of wffs.

$L \vdash A$ denotes a theorem stating that A follows from the assumptions in L.

$\dfrac{L_1 \vdash A_1 \quad \cdots \quad L_n \vdash A_n}{L \vdash A}$ is an inference rule which says that if $L_1 \vdash A_1 \cdots L_n \vdash A_n$ are theorems then we are permitted in infer that $L \vdash A$ is a theorem.

In proofs (see below) we normally record the assumption list by writing down the line numbers on which the assumptions were introduced.

1. Rule of assumption (A): $\dfrac{}{\{A\} \vdash A}$

   Any assumption may be introduced at any stage of the proof, but of course we have to keep track of which conclusions depend on which assumptions.

2. Modus ponendo ponens (MPP): $\dfrac{L \vdash A \qquad L' \vdash A \Rightarrow B}{L \cup L' \vdash B}$

3. Modus tollendo tollens (MTT): $\dfrac{L \vdash \neg B \qquad L' \vdash A \Rightarrow B}{L \cup L' \vdash \neg A}$

4. Conditional proof (CP): $\dfrac{L \cup \{A\} \vdash B}{L \vdash A \Rightarrow B}$

   Given a proof of B from some assumptions including A, we can get a proof of $A \Rightarrow B$ from the remaining assumptions.

5. Double Negation (DN): $\dfrac{L \vdash A}{L \vdash \neg\neg A}$ and $\dfrac{L \vdash \neg\neg A}{L \vdash A}$

6. $\wedge$-Introduction ($\wedge$I): $\dfrac{L \vdash A \quad L' \vdash B}{L \cup L' \vdash A \wedge B}$

7. $\wedge$-Elimination ($\wedge$E): $\dfrac{L \vdash A \wedge B}{L \vdash A}$ and $\dfrac{L \vdash A \wedge B}{L \vdash B}$

8. $\vee$-Introduction ($\vee$I): $\dfrac{L \vdash A}{L \vdash A \vee B}$ and $\dfrac{L \vdash B}{L \vdash A \vee B}$

9. $\vee$-Elimination ($\vee$E): $\dfrac{L \vdash A \vee B \quad L' \cup \{A\} \vdash C \quad L'' \cup \{B\} \vdash C}{L \cup L' \cup L'' \vdash C}$

If $L \vdash A \vee B$ then if all the assumptions in L are true, either A is true or B is true. $L' \cup \{A\} \vdash C$ is a proof of C (from some additional assumptions) assuming that A is true. $L'' \cup \{B\} \vdash C$ is a proof of C (from some additional assumptions, possibly different) assuming that B is true.

10. Reductio ad absurdum (RAA): $\dfrac{L \cup \{A\} \vdash B \wedge \neg B}{L \vdash \neg A}$

If we can prove $B \wedge \neg B$ from $L \cup \{A\}$ then the assumptions $L \cup \{A\}$ are inconsistent.

See Lemmon for explanations of all these rules.

Some example proofs

A proof that $\vdash (P \Rightarrow Q) \Rightarrow ((P \Rightarrow \neg Q) \Rightarrow \neg P)$:

Assumptions	Line no.	Conclusion	Inference Rule
1	1	$P \Rightarrow Q$	A
2	2	$P \Rightarrow \neg Q$	A
3	3	P	A
1,3	4	Q	1,3 MPP
2,3	5	$\neg Q$	2,3 MPP
1,2,3	6	$Q \wedge \neg Q$	4,5 $\wedge$I
1,2	7	$\neg P$	3,6 RAA
1	8	$(P \Rightarrow \neg Q) \Rightarrow \neg P$	2,7 CP
	9	$(P \Rightarrow Q) \Rightarrow ((P \Rightarrow \neg Q) \Rightarrow \neg P)$	1,8 CP

Lines 1-8 are a proof that $P \Rightarrow Q \vdash (P \Rightarrow \neg Q) \Rightarrow \neg P$.

A proof that $\vdash (P \Rightarrow \neg P) \Rightarrow \neg P$:

Assumptions	Line no.	Conclusion	Inference Rule
1	1	$P \Rightarrow \neg P$	A
2	2	P	A
1,2	3	$\neg P$	1,2 MPP
1,2	4	$P \wedge \neg P$	2,3 $\wedge$I
1	5	$\neg P$	2,4 RAA
	6	$(P \Rightarrow \neg P) \Rightarrow \neg P$	1,5 CP

See Lemmon for many more examples.

First-order logic

First-order logic can be regarded as an extension of propositional logic. It allows us to handle individuals (like people and things) and generalization in a systematic way.

Reference: Lemmon

Well-formed formulae

There are two things we have to do to extend propositional logic to first-order logic.

1. Change the notion of atomic formula

Previously we said that any atomic proposition (i.e. propositional variable) was a wff. We now introduce the notion of a predicate (a property which an individual can have or which relates a number of individuals). We also need constants (which represent particular individuals), variables (which represent arbitrary individuals), and functions which map tuples of individuals to an individual.

A term is either:
- a constant c,
- a variable x, or
- $f(t_1, \ldots, t_n)$ where f is a function and $t_1, \ldots, t_n$ are terms.

An atomic formula has the form $P(t_1, \ldots, t_n)$, where P is a predicate and $t_1, \ldots, t_n$ are terms.

Example:

even(times(2,x)) is an atomic formula
2 is a constant
x is a variable
times is a function
2, x and times(2,x) are terms
even is a predicate

2. Add quantifiers $\forall$ and $\exists$

So a <u>well-formed formula</u> is either:

- an atomic formula,
- an expression of the form
  $P \wedge Q$, $P \vee Q$, $\neg P$, or $P \Rightarrow Q$
  where P, Q are wffs, or
- an expression of the form
  $\forall x.P$ or $\exists x.P$
  where x is a variable and P is a wff.

Note: Lemmon uses the notation $(x)P$ instead of $\forall x.P$

Example:

$\forall x.\ F(x) \Rightarrow G(x)$	(all Frenchmen are generous)
$\exists x.\ F(x) \wedge \neg G(x)$	(some Frenchmen are not generous)
$\forall x.\exists y.\ P(x,y)$	(everyone has a parent)

All the stuff about propositional logic remains valid, e.g. the meanings of the connectives and the inference rules. Using truth tables to determine if a wff is a tautology no longer makes sense, and we need some additional inference rules to handle $\forall$ and $\exists$.

Translating common forms of sentences into predicate logic

Everything with F has G: $\forall x.\, F(x) \Rightarrow G(x)$
Something with F has G: $\exists x.\, F(x) \wedge G(x)$
Nothing with F has G: $\forall x.\, F(x) \Rightarrow \neg G(x)$
Something with F has not G: $\exists x.\, F(x) \wedge \neg G(x)$

Inference rules

We need four additional inference rules to reason about quantifiers. Let A and B be wffs, Let c be a constant and let x be a variable. The notation $A[c/x]$ denotes the result of substituting c for each unbound occurrence of x in A. The word "unbound" is important: for example, $(P(x) \wedge \forall x.\, Q(x))[c/x]$ is $P(c) \wedge \forall x.\, Q(x)$, not $P(c) \wedge \forall x.\, Q(c)$ and especially not $P(c) \wedge \forall c.\, Q(c)$.

1. Universal elimination ($\forall E$): $$\frac{L \vdash \forall x.\, A}{L \vdash A[c/x]}$$

This is like a generalization of $\wedge E$, since you can think of $\forall x.\, P(x)$ as an abbreviation for $P(c_1) \wedge \cdots \wedge P(c_n)$ where $c_1, \ldots, c_n$ are all the individuals in the universe.

2. Universal introduction ($\forall I$): $$\frac{L \vdash A[c/x]}{L \vdash \forall x.\, A}$$
provided that c does not appear in L

This is (sort of) like a generalization of $\wedge I$. The reason for the side condition (provided that...) will be explained below.

3. Existential Elimination ($\exists E$): $\dfrac{L \vdash \exists x.A \quad L' \cup \{A[c/x]\} \vdash B}{L \cup L' \vdash B}$

provided that c does not appear in L' or B

This is (sort of) like a generalization of $\vee E$, since you can think of $\exists x.P(x)$ as an abbreviation for $P(c_1) \vee \dots \vee P(c_n)$ where $c_1, \dots, c_n$ are all the individuals in the universe. The reason for the side condition will be explained below.

4. Existential Introduction ($\exists I$) $\dfrac{L \vdash A[c/x]}{L \vdash \exists x.A}$

This is like a generalization of $\vee I$.

Why the side conditions on $\forall I$ and $\exists E$?

The side conditions on $\forall I$ and $\exists E$ are to ensure that the constant c denotes an arbitrary individual.

Let's try dropping these conditions — then it is possible to prove statements which are not true.

A "proof" that all Americans are millionaires:

1	1. M(Prince Charles)	A
2	2. A(Prince Charles)	A
1,2	3. M(Prince Charles) $\wedge$ A(Prince Charles)	1,2 $\wedge$I
1,2	4. M(Prince Charles)	3 $\wedge$E
1	5. A(Prince Charles) $\Rightarrow$ M(Prince Charles)	2,4 CP
1	6. $\forall x.(A(x) \Rightarrow M(x))$	5 $\forall$I

In this example, $M(\text{PrinceCharles})$ represents the statement that PrinceCharles is a millionaire, and $A(\text{PrinceCharles})$ represents the statement that PrinceCharles is an American.

Steps 3 and 4 are necessary just to get $A(\text{PrinceCharles})$ as an assumption of $M(\text{PrinceCharles})$ so that CP can be applied in step 5.

The "proof" says $M(\text{PrinceCharles}) \vdash \forall x.\,(A(x) \Rightarrow M(x))$. Since $M(\text{PrinceCharles})$ is true), this says that all Americans are millionaires.

The problem is, of course, in step 6 — PrinceCharles appears in the assumptions on which step 5 depends, so $\forall I$ is not applicable.

A "proof" that I am a millionaire:

1	1. $\exists x.\, M(x)$	A
2	2. $M(\text{DonSannella})$	A
1	3. $M(\text{DonSannella})$	1,2,2 $\exists E$

This "proof" says that I am a millionaire under the assumption that somebody is a millionaire. The former is true but the latter is not.

The problem is in step 3 — the constant DonSannella appears in the result (this is B in the inference rule above).

A "proof" that some number is both even and odd:

1	1. $\text{even}(0)$	A
2	2. $\exists x.\, \text{odd}(x)$	A
3	3. $\text{odd}(0)$	A
1,3	4. $\text{even}(0) \wedge \text{odd}(0)$	1,3 $\wedge I$
1,3	5. $\exists x.\,(\text{even}(x) \wedge \text{odd}(x))$	4 $\exists I$
1,2	6. $\exists x.\,(\text{even}(x) \wedge \text{odd}(x))$	2,3,5 $\exists E$

This says that some number is both even and odd, provided that 0 is even (which is true) and there exists an odd number (which is true).

The problem is in step 6 — step 5 rests on the assumption even(0) which contains the constant 0, so ∃E is not applicable.

Soundness and completeness

As with propositional logic, first-order logic is both
- sound (everything I can prove is true), and
- complete (everything which is true can be proved).

But in propositional logic it is possible to construct a proof of any true formula mechanically (The basic idea is to construct a truth table and then convert that into a proof). In predicate calculus this is not possible.

That is, the set of theorems of propositional logic is recursive while the set of theorems of first-order logic is only recursively enumerable. (You should have met these terms in CS3 computability but if you haven't it doesn't matter).

For example proofs see Lemmon.

Resolution theorem proving

Reference:
A. Bundy, The Computer Modelling of Mathematical Reasoning, Academic Press, 1984.

Motivation

Doing proofs in first-order logic is hard. It is hard to choose the right rule to apply next; we also have to choose how to apply it to the proof we have constructed so far (for example, when applying Conditional Proof we have to decide which assumption to discharge). This means that automatic theorem proving is a hard search problem.

Automatic theorem proving is important for knowledge representation (among other things) so we would like it to be easier.

One approach is to adopt a resolution method. The idea is to replace all the inference rules (there are 17 if we count DN, $\wedge$E and $\vee$I as two rules each) with just 1 rule of which they are all a special case. Then the problem of choosing the right inference rule to apply at each step goes away, although there is still the problem of choosing how to apply it.

Before discussing resolution it is necessary to explain how every wff of first-order logic can be converted to a special form called conjunctive normal form. Resolution only works on formulae of this form. Before discussing conjunctive normal form it is necessary to explain Skolemization which is about getting rid of existential quantifiers.

Skolemization

Consider the formula:

$$\exists x.\ (man(x) \wedge big(x))$$

This is equivalent to: $man(bigman) \wedge big(bigman)$ *provided* bigman is a constant which has been unused up to now. After all, "bigman" is just being used as a *name* for the x such that $man(x) \wedge big(x)$. As long as this name has not been used yet there is no danger of confusion.

So, this suggests that we can replace existentially-bound variables with appropriately chosen constants. These are called *Skolem constants*, after Thoralf Skolem, the logician who invented this trick.

This trick doesn't always work, though; consider the formula:

$$\forall x.\ (boy(x) \Rightarrow \exists y.\ (girl(y) \wedge loves(x,y)))$$

(every boy loves a girl). This is *not* equivalent to

$$\forall x.\ (boy(x) \Rightarrow (girl(Jane) \wedge loves(x, Jane)))$$

which means that every boy loves the *same* girl. We have to make the constant "Jane" *depend* on *the value* of x.

So we can use a Skolem *function*, girlfriend:

$$\forall x.\ (boy(x) \Rightarrow (girl(girlfriend(x)) \wedge loves(x, girlfriend(x))))$$

again, provided that girlfriend is an unused function symbol.

In general, these new functions may need several arguments, one for each universal quantifier in whose scope they fall. For example, $\forall x.\forall y.\ (\neg eq(x,y) \Rightarrow \exists z.\ (lt(x,z) \wedge lt(z,y)))$ (i.e. for every two nonequal numbers there is another number between them) is equivalent to

$$\forall x.\forall y.\ (\neg eq(x,y) \Rightarrow (lt(x, between(x,y)) \wedge lt(between(x,y), y)))$$

Conjunctive Normal Form (CNF)

A formula is in CNF if it consists of a conjunction of disjunctions

$$(A_1 \vee \cdots \vee A_n) \wedge (B_1 \vee \cdots \vee B_m) \wedge \cdots$$

where all the A_j's, B_j's etc are <u>literals</u>, i.e. either atomic formulae or negated atomic formulae. Any variable in such a formula is implicitly universally quantified, that is

$$(A_1 \vee \cdots \vee A_n) \wedge (B_1 \vee \cdots \vee B_m) \wedge \cdots$$

is equivalent to the non-CNF formula

$$\forall x_1\, \forall x_2 \cdots \forall x_p .\, [\,(A_1 \vee \cdots \vee A_n) \wedge (B_1 \vee \cdots \vee B_m) \wedge \cdots\,]$$

where $x_1 \cdots x_p$ are all the variables in $A_1, \ldots, A_n, B_1, \ldots, B_m, \ldots$.

Every wff can be converted to an equivalent formula in CNF by the following procedure:

1. Get rid of $\Rightarrow$ using the identity $A \Rightarrow B = (\neg A) \vee B$
2. Push $\neg$'s innermost using
   $\neg\neg A = A$
   $\neg(A \wedge B) = \neg A \vee \neg B$ $\quad$ $\neg(A \vee B) = \neg A \wedge \neg B$
   $\neg \forall x.A = \exists x.\neg A$ $\quad$ $\neg \exists x.A = \forall x.\neg A$
3. Rename variables to make names unique
4. Forget $\exists$'s using Skolemization
5. Forget $\forall$'s — all variables are implicitly universally quantified
6. Apply $(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$
   and $C \vee (A \wedge B) = (C \vee A) \wedge (C \vee B)$ repeatedly
   (as left-to-right rewrite rules).

Example:

$\forall x.\ (boy(x) \Rightarrow \exists y.\ (girl(y) \wedge loves(x,y)))$

1. $= \forall x.\ (\neg boy(x) \vee \exists y.\ (girl(y) \wedge loves(x,y)))$
2. no change
3. no change
4. $= \forall x.\ (\neg boy(x) \vee (girl(girlfriend(x)) \wedge loves(x, girlfriend(x))))$
5. $= \neg boy(x) \vee (girl(girlfriend(x)) \wedge loves(x, girlfriend(x)))$
6. $= (\neg boy(x) \vee girl(girlfriend(x))) \wedge (\neg boy(x) \vee loves(x, girlfriend(x)))$

Example: $\forall x.\ (S(x) \Rightarrow \neg (\exists y.\ (P(y) \Rightarrow Q(x)) \Rightarrow \forall x.\ R(x)))$

1. $= \forall x.\ (\neg S(x) \vee \neg (\neg \exists y.\ (\neg P(y) \vee Q(x)) \vee \forall x.\ R(x)))$
2. $= \forall x.\ (\neg S(x) \vee (\exists y.\ (\neg P(y) \vee Q(x)) \wedge \exists x.\ \neg R(x)))$
3. $= \forall x.\ (\neg S(x) \vee (\exists y.\ (\neg P(y) \vee Q(x)) \wedge \exists z.\ \neg R(z)))$
4. $= \forall x.\ (\neg S(x) \vee ((\neg P(f(x)) \vee Q(x)) \wedge \neg R(g(x))))$
5. $= \neg S(x) \vee ((\neg P(f(x)) \vee Q(x)) \wedge \neg R(g(x)))$
6. $= (\neg S(x) \vee \neg P(f(x)) \vee Q(x)) \wedge (\neg S(x) \vee \neg R(g(x)))$

The procedure for converting wffs to CNF can be used for propositional logic if we just omit steps 3–5.

Interesting fact: In propositional logic, A is a tautology if and only if each conjunct in CNF(A) contains some propositional variable and its negation. So we can use this process as an alternative to truth table for tautology checking.

Example: $(P \Rightarrow Q) \Rightarrow ((P \Rightarrow \neg Q) \Rightarrow \neg P)$

converts to: $(P \vee \neg P) \wedge (\neg Q \vee P \vee \neg P) \wedge (P \vee Q \vee \neg P) \wedge (\neg Q \vee Q \vee \neg P)$

Each conjunct contains either both P and $\neg P$ or both Q and $\neg Q$ so this formula is a tautology.

Resolution

The reason why we bother turning wffs into CNF is that resolution only works on formulae in that form. A formula $(A_1 \vee \dots \vee A_n) \wedge (B_1 \vee \dots \vee B_m) \wedge \dots$ is equivalent to a set of disjunctions $A_1 \vee \dots \vee A_n$, $B_1 \vee \dots \vee B_m$, etc which are all required to be true. Resolution deals with sets of disjunctions (clauses), telling you how to put two disjunctions together to get a new one which can be added to the set.

Resolution will be explained in 3 stages, starting with the simplest case and working up to the most general form of the inference rule.

1. Variable-free resolution $\qquad \dfrac{C \vee P \qquad C' \vee \neg P}{C \vee C'}$

$\neg P, P$ are called complementary literals. A truth table shows that this rule is valid:

P	C	C'	C∨P	C'∨¬P	C∨C'
T	T	T	T	T	T
T	T	F	T	F	T
T	F	T	T	T	T
T	F	F	T	F	F
F	T	T	T	T	T
F	T	F	T	T	T
F	F	T	F	T	T
F	F	F	F	T	F

Whenever both $C \vee P$ and $C' \vee \neg P$ is T, so is $C \vee C'$

By the way, since $\vee$ is commutative, $C \vee P = P \vee C$ and $C' \vee \neg P = \neg P \vee C'$ so the order of literals above is unimportant.

A special case of variable-free resolution is the MPP rule:

$$\frac{A \Rightarrow B \qquad A}{B}$$

Since $A \Rightarrow B = \neg A \vee B$, this is just

$$\frac{\neg A \vee B \qquad A}{B}$$

which is the same as the rule above, where $P = A$, $C = F$ and $C' = B$

2. Binary resolution

The two clauses might not contain complementary literals, but it might be possible to make them complimentary by applying a substitution (i.e. taking a special case by substituting terms for variables):

$$\frac{C \vee P \qquad C' \vee \neg P'}{(C \vee C')\varphi}$$

where φ is the most general unifier of P, P'
i.e. $\varphi(P) = \varphi(P')$

3. Full resolution

It may be possible to eliminate not just one but several literals simultaneously.

$$\frac{C \vee P_1 \vee \cdots \vee P_m \qquad C' \vee \neg P_1' \vee \ldots \vee \neg P_n'}{(C \vee C')\varphi}$$

where φ is the most general unifier of all the P's and P's

The resolution method

Proof by resolution proceeds as follows:

1. Turn premises into CNF. Result: set of clauses.
2. Negate the conclusion; turn it into CNF. Result: set of clauses.
3. Take union of 1 and 2
4. Apply resolution rule until the empty clause is obtained (the empty clause is a disjunction containing no literals)

This amounts to a proof by reductio ad absurdum of the conclusion from the premises since the empty clause is equivalent to F.

Resolution is complete for first order logic.

Paramodulation is a related technique used to speed up reasoning about equality. The rule is:

$$\frac{C(T) \qquad T=S}{C(S)}$$

Logic for knowledge representation — recap

We've covered propositional logic (very simple but not too useful) and first-order logic (quite useful for encoding every-day factual knowledge) Theorem proving is by means of inference rules, and it is possible to speed up theorem proving using the resolution method.

So the idea is that the knowledge base contains a bunch of first-order formulae, and theorem proving is used to infer facts which are implicit in the knowledge.

Some problems with logic for knowledge representation

You are probably thinking that this scheme is all very nice and orderly but somehow it isn't plausible that knowledge is *really* represented in the form of formulae in our heads, and clearly we are *not* doing theorem proving when we make connections between things that we know (for one thing, we make mistakes).

Logic is also not good at representing meta-knowledge, procedural knowledge or probabilistic knowledge, or the idea that objects in a certain class have default properties which can be overridden on occasion (e.g., birds can fly except penguins can't). These problems can be overcome but the solutions are more or less clumsy.

Another thing — if all knowledge is represented as a bunch of facts, it would seem to be difficult to locate anything. If we want to know what colour a table is we seem to have to search through a big database for a fact of the form colour(table, ...) which seems overly difficult. It could be argued that we should just know it immediately, i.e. it should be directly connected with the other knowledge we have about that table. In fact, this isn't a terribly valid criticisms of logic — there are sophisticated indexing schemes which make it possible to find the right facts fairly fast.

There are other knowledge representation schemes which seem (at least superficially) more attractive.

Introduction to semantic networks

There is no fixed definition of what constitutes a semantic network. A large number of quite different schemes are grouped together under this heading. The main thing they have in common is that they use graphs of some form or other to represent knowledge. So nodes typically represent objects or classes of objects, while arcs (links, arrows) represent relations between them.

(There is a close relationship between this and graphical notations for database schemes.)

The basic ideas of one sort of semantic network

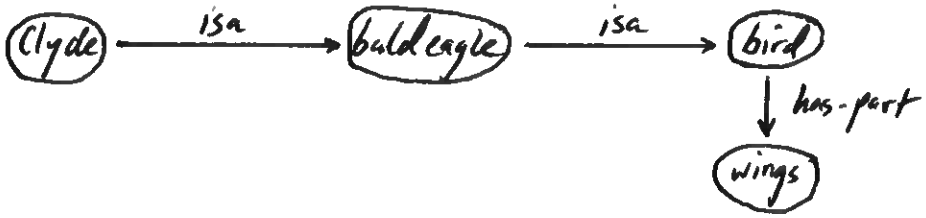
"All bald eagles are birds" can be represented by creating two nodes to represent bald eagles and birds with a link connecting them:

If Clyde is a particular individual who we wish to assert is a bald eagle, we could add a node for Clyde to the network:

This not only represents the two facts we intended, but also it has made it easy to deduce another fact, namely that Clyde is a bird, just by following the isa links: Clyde is a bald eagle, bald eagles are birds; hence Clyde is a bird.

This is one attraction of this sort of representation scheme: it is very easy to make certain kinds of inferences. This is especially appropriate in a domain where much of the reasoning needed is based on a complicated taxonomy or classification scheme. An example is the PROSPECTOR system which includes a taxonomical classification of different kinds of rocks. If the user says he sees iron pyrites, the system triggers rules about those, but since iron pyrites are sulphides it triggers rules about those too.

We usually want to represent more than just a classification of objects; we want to represent knowledge about their properties as well. For example, "birds have wings" can be added to the previous example:



(the direction that the arrows point is not all that important, as long as we are consistent:

would obviously be wrong.)

Again, the representation makes it easy to deduce that bald eagles have wings and that Clyde has wings.

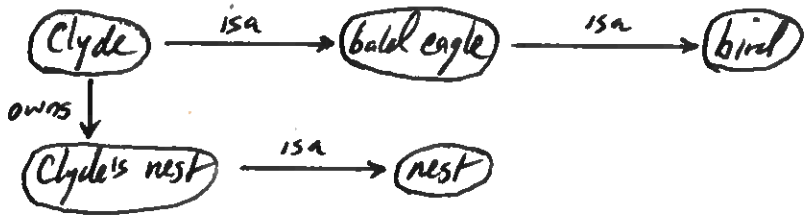
All you have to do is look up the isa-hierarchy, taking any assumptions about higher nodes as assertions about lower ones as well without having to represent those assertions explicitly in the net. This is called property inheritance.

This idea gives what is sometimes known as cognitive economy. We want to represent properties of concepts in the most economical way. This means that we want to attach a property like "has wings" as high up in the isa-hierarchy as possible, rather than repeatedly in different places lower down.

There are similar ideas in object-oriented programming languages like SMALLTALK.

Situations

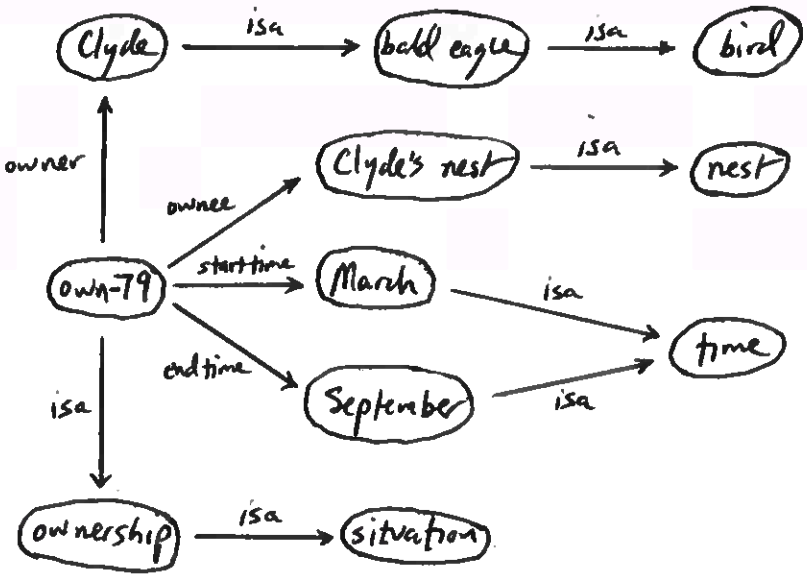
Suppose we want to represent the fact that "Clyde owns a nest". We might encode this using an ownership link between Clyde and his nest:



But this isn't quite right, since Clyde only owns a nest from March until September. In logic we would probably use a 4-argument predicate:

owns(owner, ownee, start time, end time)

The use of arrows in semantic nets gives us only binary predicates. A solution is to allow nodes to represent situations and actions as well as objects and sets of objects. Each situation node can have a set of outgoing arcs called a case frame which are always grouped together. For this example we need a case frame with 4 outgoing arcs (plus one saying that this is a case frame):



Own-79 knows that it is supposed to have these 4 arcs because it isa ownership.

One important use of this is that we can arrange things so that nodes like own79 inherit default values for certain of their properties. For example, the default values for start time and end time for the situation of ownership might be the beginning and end of the owner's life, respectively. (Here ownership has to compute a default value depending on who the owner is in a particular instance. This kind of complex cross-reference is not easy to arrange with semantic nets because there isn't really anywhere to hang the required bits of code.)

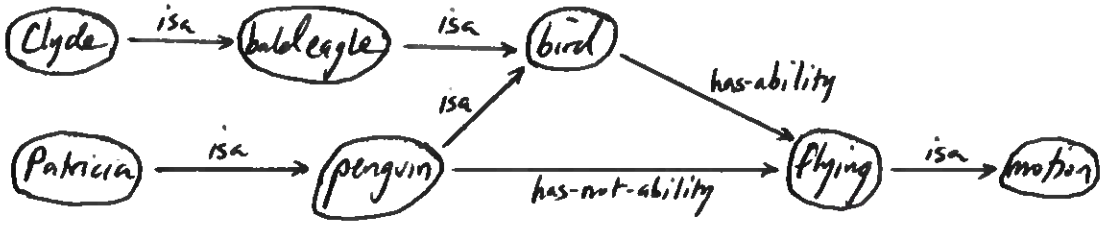
Defaults and overriding them

Semantic networks make it easy to arrange for properties of a class to be inherited by instances of that class.

Sometimes we want these inherited properties to be regarded as default values which can be overridden (as in the above example with default values for start time and end time).

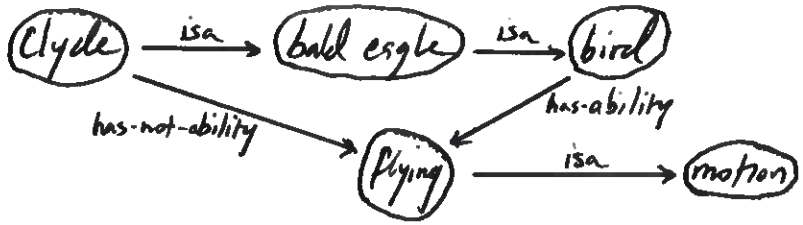
For example, birds can fly:

So we know that bald eagles, for example Clyde, can fly. But penguins can't fly, so we need to override the default:



(has-not-ability is another hack — we really need proper negation)

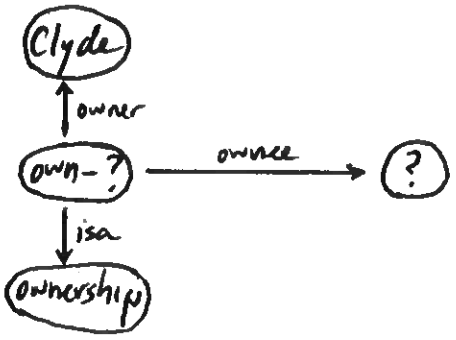
We can use the same idea to represent the fact that Clyde can't fly although he is a bald eagle (maybe he has a broken wing):



Inference by matching

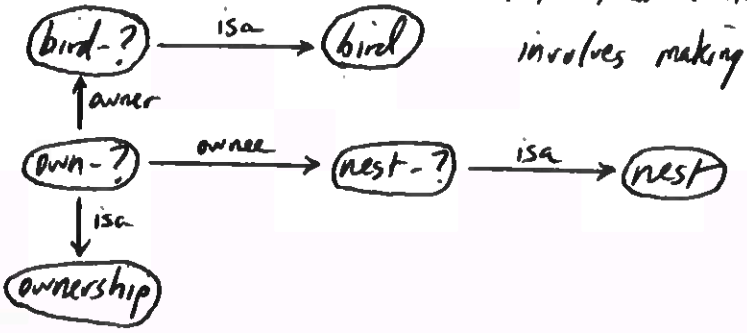
One useful reasoning mechanism is based on matching network structures. A network fragment is constructed, representing a sought-for object or a query, and then matched against the network database to see if such an object exists. Variable nodes in the fragment are bound in the matching process to the values they must have to make the match perfect.

For example, suppose we have a database containing the networks above including the one asserting that Clyde owns a nest between March and September. Suppose we want to answer the question "What does Clyde own?" We construct the fragment:



This fragment is matched against the network database looking for an own node that has an owner link to Clyde. When it is found, the node that the ownee link points to is bound in the partial match and is the answer to the question.

A more complicated example: "Is there a bird who owns a nest?"



Note that matching of this fragment involves making an inference (Clyde is a bird.)

Semantic networks — a survey

Reference:
R.J. Brachman, "On the epistemological status of semantic networks" in Brachman + Levesque Readings in Knowledge Representation.

This is a survey of some work involving semantic networks. You are not expected to learn the details of the different notations, etc., just to have a general understanding of what people have done with semantic networks.

Quillian

Reference:
M.R. Quillian, "Word concepts: a theory and simulation of some basic semantic capabilities" in Brachman + Levesque.

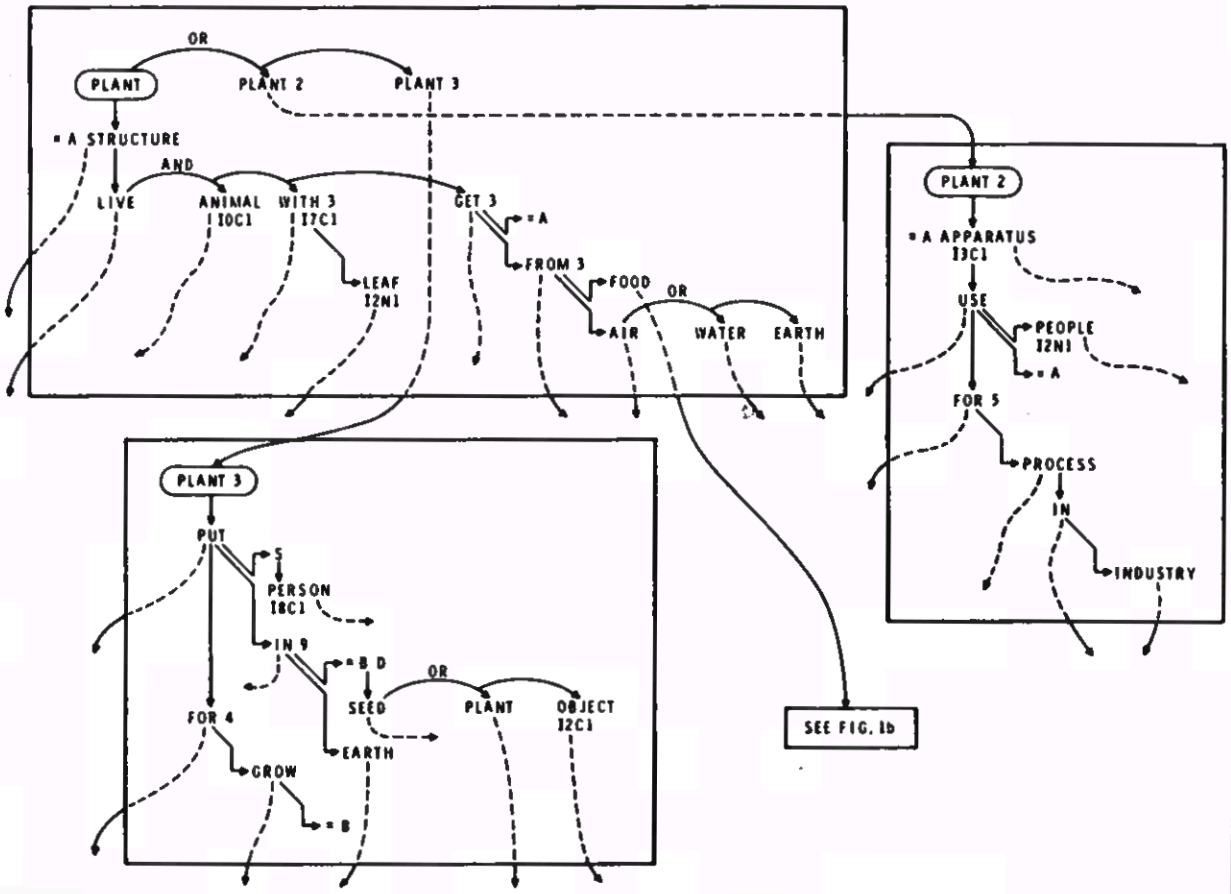
Quillian introduced the idea of semantic networks in 1966. He used networks for representing the semantics of English words. The representation closely reflects the structure of an ordinary dictionary, with words defined by reference to other words. (See examples below).

The idea is that each word (or word concept) is defined by a network which is put together using pointers to other word concept networks and bits of primitive structure like conjunction, disjunction, modification, and subject/object relation. Such a word definition is called a plane.

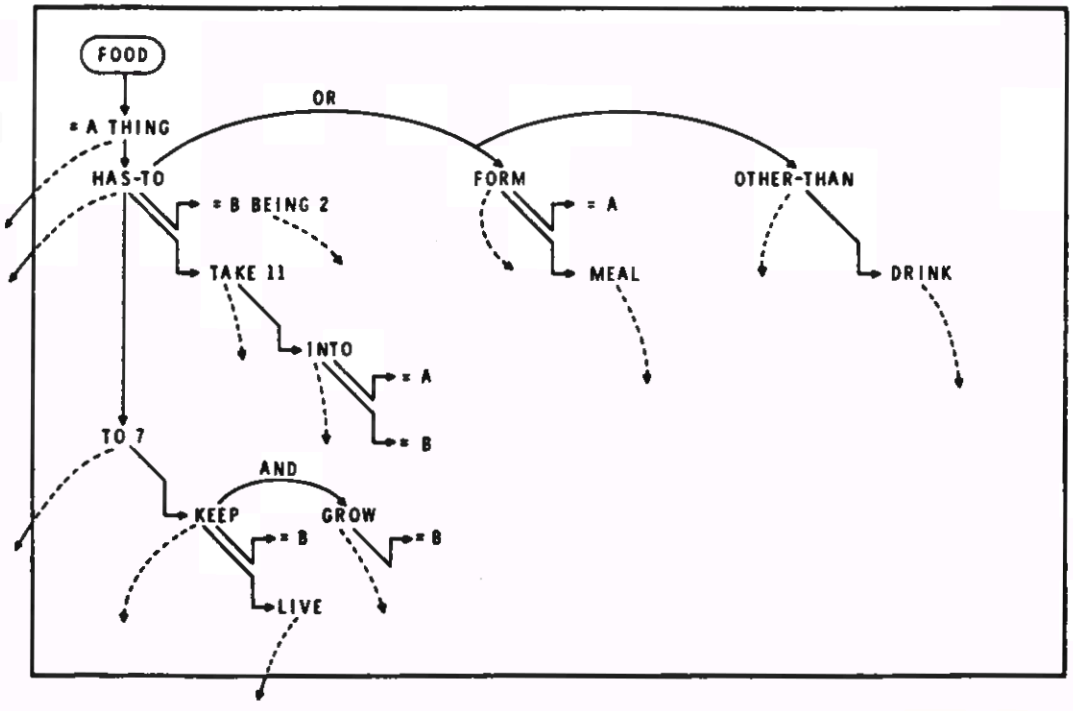
PLANT. 1. Living structure which is not an animal, frequently with leaves, getting its food from air, water, earth.

2. Apparatus used for any process in industry.

3. Put (seed, plant, etc.) in earth for growth.

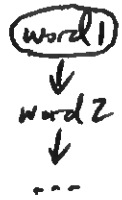


FOOD: 1. That which living being has to take in to keep it living and for growth. Things forming meals, especially other than drink



Compare the networks with the corresponding dictionary entries.

=A is like a pronoun
with 3 refers to the third sense of "with"
10C1 etc. are modifiers (not, usually, sometimes etc.)
S, D (in plant 3) are parameters — placeholders for person, thing etc. which can be instantiated when the concept is used in context.



means word1 <u>isa</u> word2 such that ...

Quillian invented the so-called <u>type</u> / <u>token</u> <u>distinction</u> which you will find referred to elsewhere. A <u>type</u> is the word being defined in the plane under consideration, and <u>tokens</u> are all the other nodes in the plane. These are basically place holders for words used in the definition, each having a pointer to its associated type node which is at the "head" of another plane. In the picture, type nodes are in circles.

This just means that definitions are not duplicated when they are referred to.

The idea seems to be merely a complicated way of saying that pointers are used to avoid copying large chunks of structure, but Quillian spent some effort justifying it in psychological terms by talking about avoiding redundancy in the human brain, etc.

It isn't quite clear how synonyms work, or in fact what the difference is between a word like "plant" and the concept of a plant (of which there are 3). Are there separate type nodes for synonyms or not? Probably not but Quillian doesn't address this point.

Quillian defines a <u>full word concept</u> as including all the nodes you can get to starting at the type node of the word you are interested in and following all the pointers; the result is an enormous hierarchical structure which extends in lots of complicated ways throughout a large chunk of the overall network, quite possibly very circular.

No concepts are viewed as primitive (except conjunction etc) — everything is defined in terms of other things in the memory.

So what do you do with such a network once you have it?

Quillian discusses an inference technique called <u>intersection search</u> which is a nice example of the sort of thing which is very natural in a semantic network yet has no real analogue in (say) logic.

The problem is to find ways of comparing and contrasting two words — finding similarities and contrasts between their meanings.

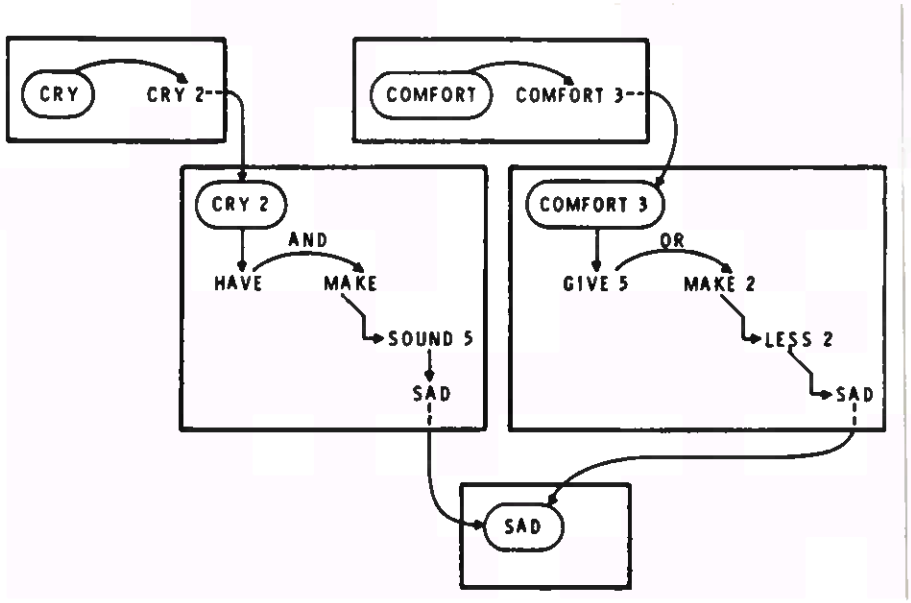
Example:

compare: cry, comfort
intersat: sad

(1) Cry 2 is among other things to make a sad sound.
(2) To comfort 3 can be to make 2 something less 2 sad.

The idea of intersection search is to simultaneously start from the two type nodes to be compared, marking all the nodes you can reach in one step from them (i.e. all the nodes in the same plane), then following all pointers and marking all the nodes you can reach from there, etc. until you reach a node which has already been marked by the other process. This is equivalent to conducting a simultaneous breadth-first search through the two full word concepts for points of contact. Quillian talks about expanding spheres of activation. A point of contact gives rise to a path back to each of the original type nodes which relates them.

Here is the example above again:



(Propositions like "for" are not taken as intersections — otherwise every pair of words would intersect.)

The sense in which this is an inference is that it demonstrates how two or more bunches of information can be put together to get relations between things.

Here,

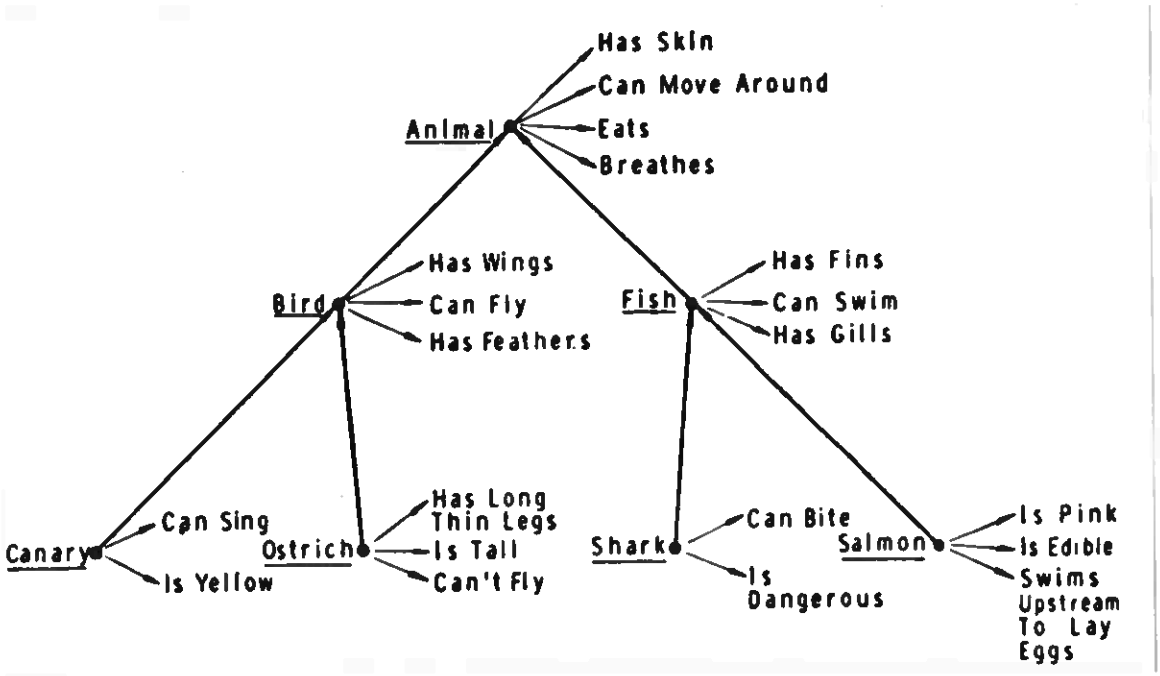
bunch of information = a word definition = a plane

A "plane-hopping" path like the one between "cry" and "comfort" represents an idea which is <u>implicit</u> in the data which was fed in.

A problem: intersection search is insufficiently constrained. This can be seen for <u>negative</u> responses — the computer has to search as far as it can go to tell that the two concepts are unrelated. This suggests that a more directed search is needed.

<u>Collins and Quillian</u>

Collins and Quillian did some experiments around 1970 to test the psychological validity of semantic networks.



They used networks like the one above — simple class hierarchies — and tested the reaction time of subjects. One would expect that it would take longer to affirm that "a canary has skin" than "a canary is yellow" since the former requires searching through two levels of the hierarchy.

The results were consistent with expectations although such experiments are never conclusive.

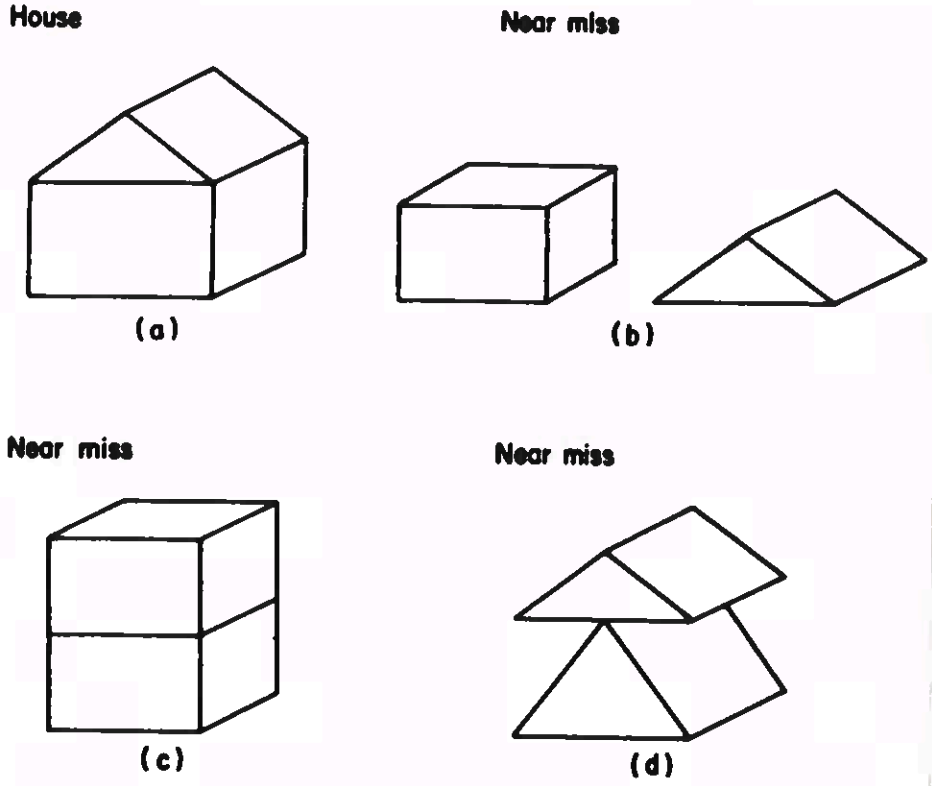
Winston

Reference:

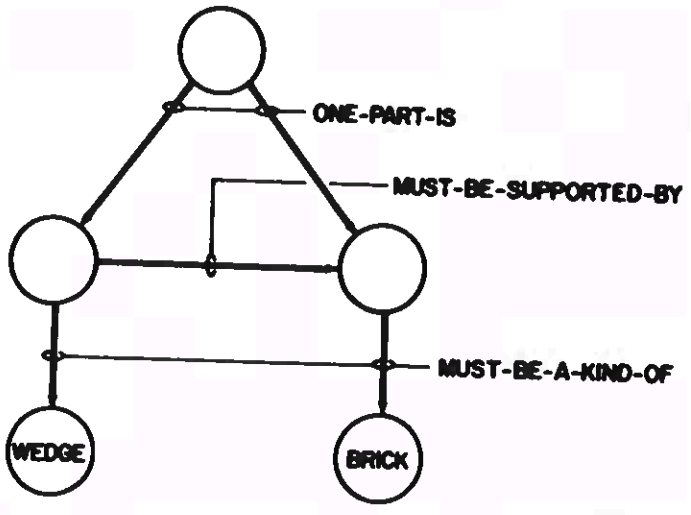
P.H. Winston, "Learning structural descriptions from examples" in Brachman + Levesque.

Winston developed a program in 1975 for learning structural descriptions of objects. This program learned the concept of an arch by being presented with a series of examples and non-examples (non-examples were supposed to be near misses, since totally arbitrary non-examples don't give any information). Each configuration could be represented by a network, and the overall concept is a network which combines features of the examples and excludes features of the non-examples, obtained by comparing the network representations.

Here is a relatively simple example: the concept of a house.

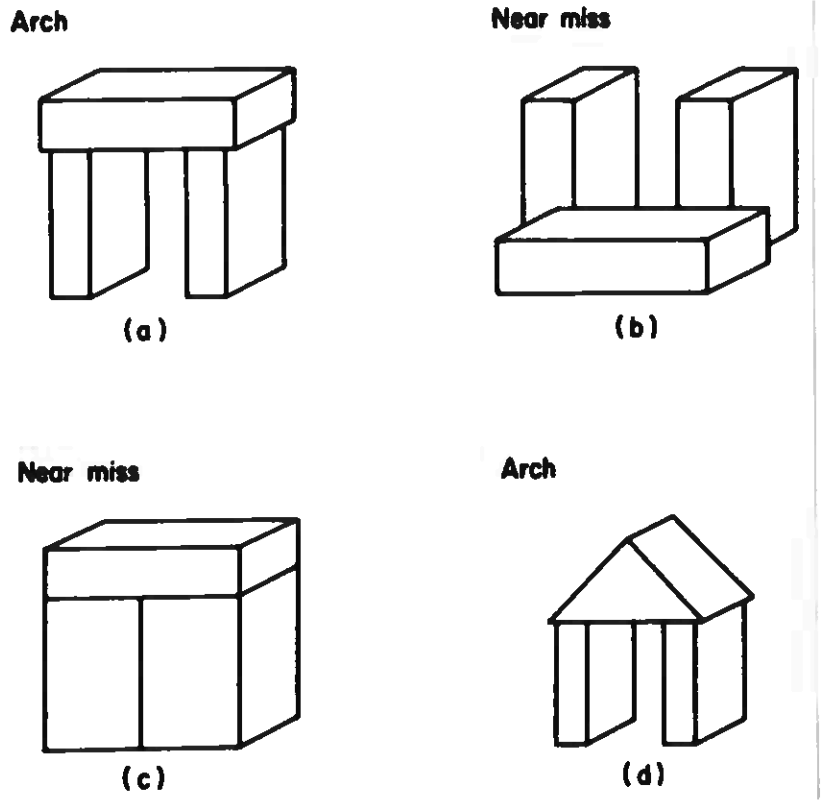


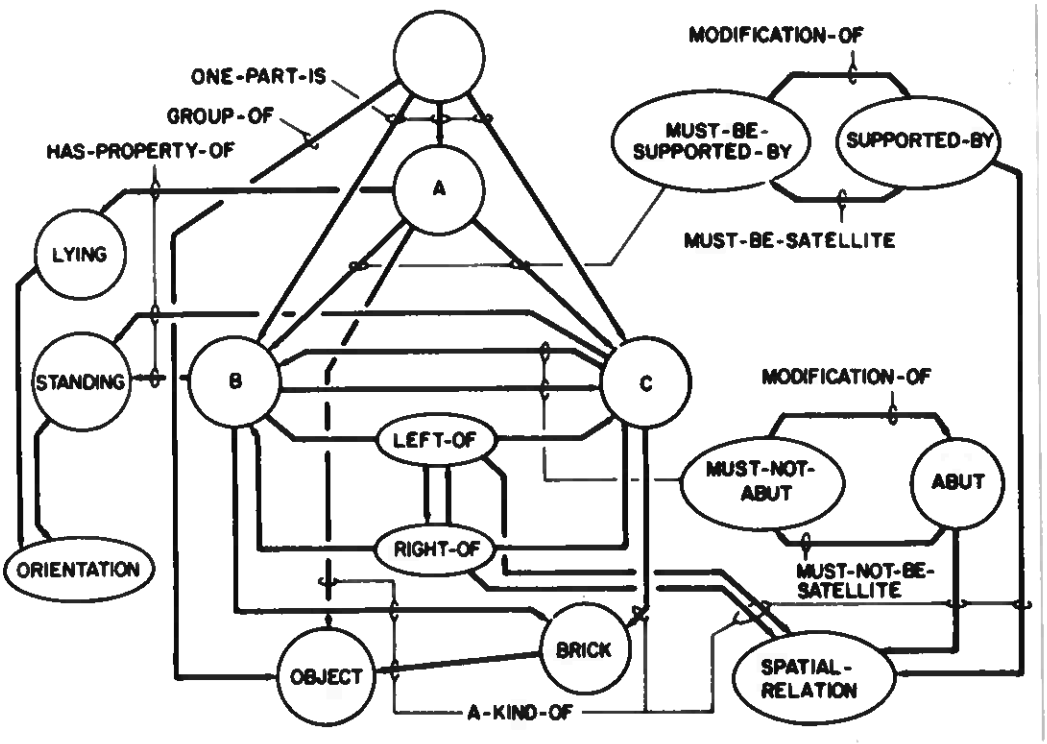
The resulting network is the following:



The arrows are just edge labels. The program knows about certain primitive things to start with: support, shapes, lying, standing, abutments, etc.

Here is a more complicated example (the concept of an arch) and the network which results.





One interesting thing about Winston's networks was that the relationships between concepts could themselves be modified or talked about as concepts. For example, the same notation can be used to describe B as left-of C and to say that left-of and right-of are spatial relations which are opposite.

Schank

Reference:

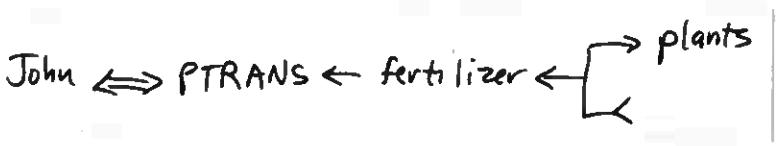
R. C. Schank and C. J. Rieger, "Inference and the computer understanding of natural language", in Brachman + Levesque.

Schank invented conceptual dependency notation for representing the meaning of sentences in natural language. The idea is to reduce everything to a small number (about 12) of primitive actions like PTRANS (physical transfer of location), INGEST (eating, drinking) etc. There are also relationships between acts like causality.

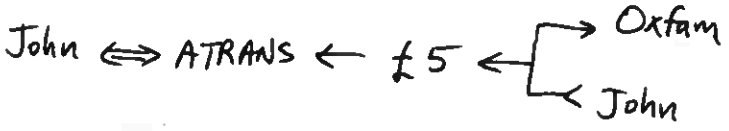
You saw conceptual dependency notation at the end of AI2 Natural Language Processing.

Here are some examples:

John put fertilizer on the plants.

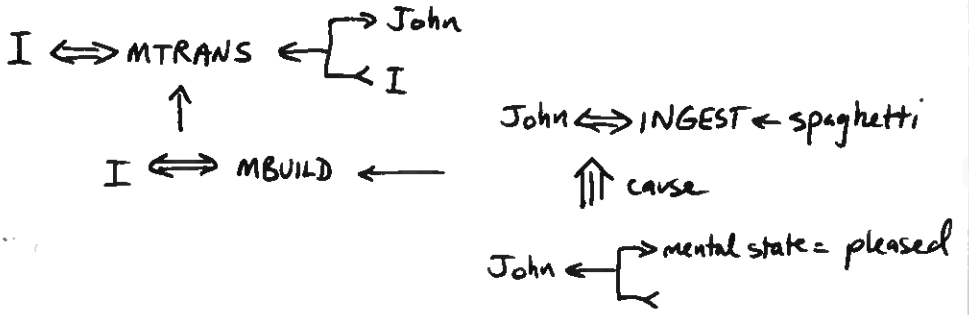


John donated £5 to Oxfam.

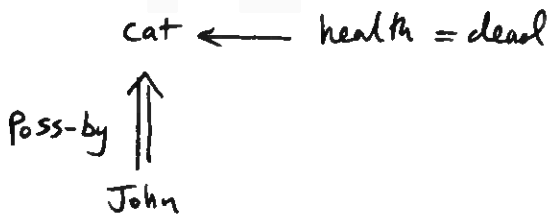


John smoked a cigar.

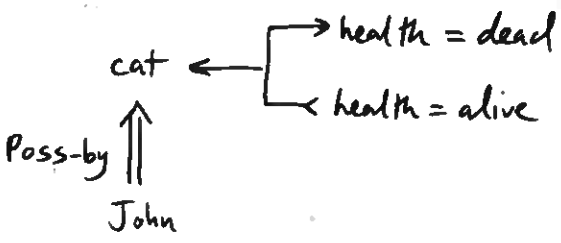
I advised John to try the spaghetti.



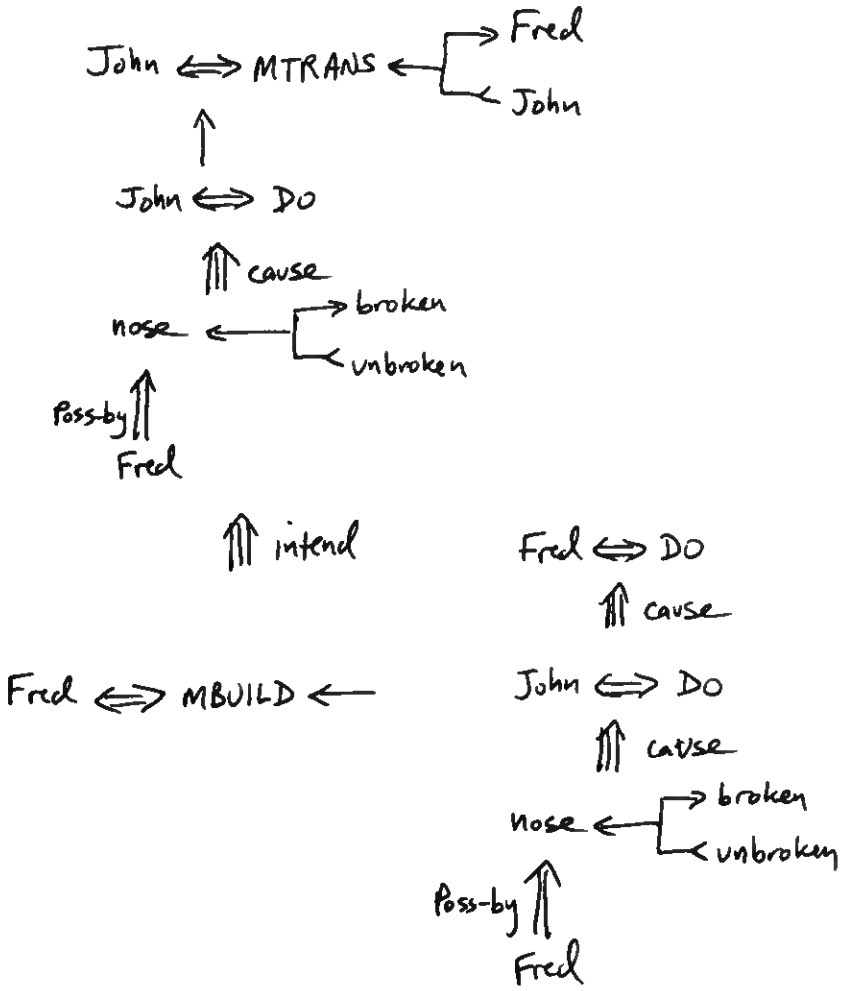
John's cat was dead.



John's cat died.



John threatened Fred with a broken nose.



The inferences which could be done with these networks included inferring that the usual result of an action took place (e.g. location changes as a result of a PTRANS) unless there is something explicitly saying that it didn't.

This work is related to Fillmore's work on case structure in linguistics (which is related to the ownership <u>case frame</u> in the last lecture).

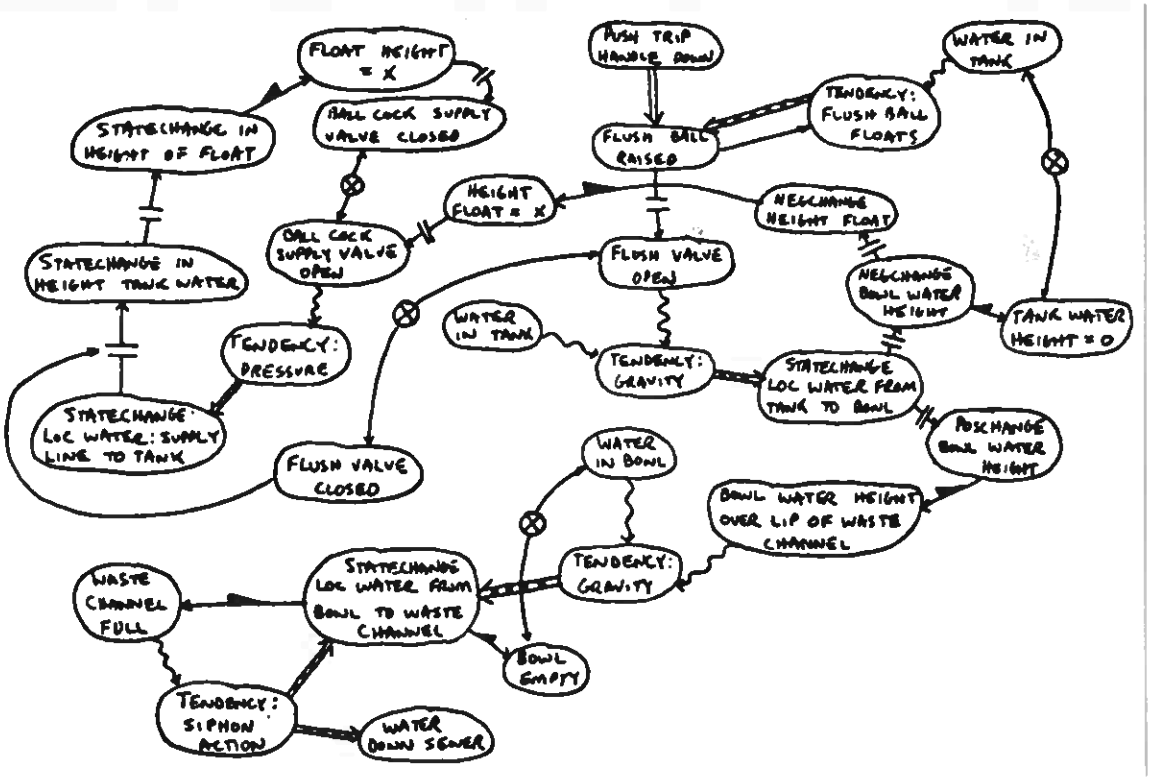
Rieger

Reference:

C. Rieger, "A organization of knowledge for problem solving and language comprehension", in Brachman + Levesque.

Rieger represented his commonsense algorithms using a network notation which shows the relationships between actions and states. There are nodes representing primitive actions like in conceptual dependency but also nodes representing states, state changes, wants and tendencies (like gravity). There is a small number (about 10) of primitive link types which are used to represent the relationships between the actions, states etc. (causality, enablement, concurrency etc.) The notation is primarily oriented toward describing how physical systems work.

Example: the operation of a reverse trap flush toilet.



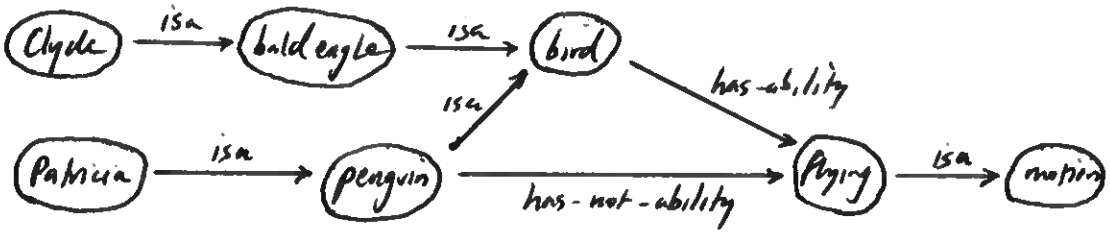
Problems with semantic networks

There are various problems with using semantic networks for knowledge representation. Two classes of problems are

- lack of flexibility
- lack of a clear semantics

Lack of flexibility: no negation

Recall the network



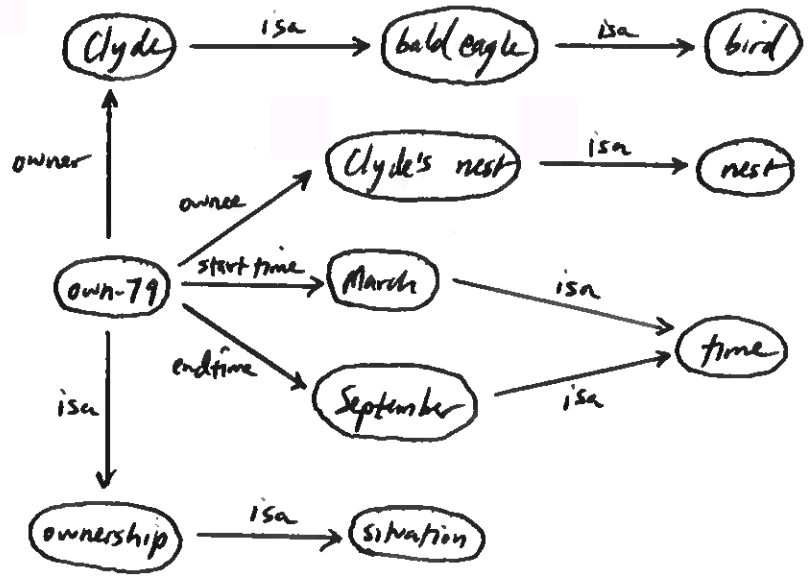
The link has-not-property is a kludge — there is no sense in which this expresses the negation of the link has-ability, at least if these are regarded as just names and not expressions with rules like those in logic about combining them to form more complex ones.

Logic is good at handling this — that's what $\neg$ is for.

Lack of flexibility: complex defaults

Remember that by attaching a property like "has-ability flying" to a node high up in the isa hierarchy, the property can be viewed as a default which is inherited by lower nodes unless it is overridden, as in the above example. But defaults sometimes have to be more complex.

Recall the example:



We would like to attach a default to ownership saying that the default start time and end time of an instance are the owner's date of birth and death respectively (to keep things simple). So, given a particular owner and ownee, we assume that start time and end time are that particular owner's birth date and death date unless told otherwise.

This is difficult to arrange because what we need is a sort of indirect reference in order to compute the default values of start time and end time.

Logic is very bad at handling defaults which can be overridden. For example, let's try the simple example of birds and flying and penguins which can't fly:

$$\forall X.(bird(X) \Rightarrow canfly(X))$$
$$\forall X.(baldeagle(X) \Rightarrow bird(X))$$
$$baldeagle(Clyde)$$
$$\forall X.(penguin(X) \Rightarrow bird(X))$$
$$\forall X.(penguin(X) \Rightarrow \neg canfly(X))$$
$$penguin(Patricia)$$

From these statements it is easy to prove

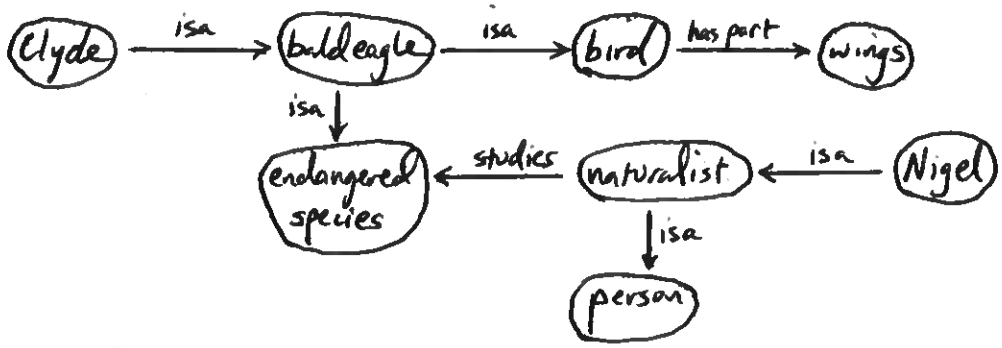
$$\text{canfly}(\text{Patricia}) \wedge \neg \text{canfly}(\text{Patricia})$$

which is a contradiction.

To handle this kind of thing we need a different sort of logic called <u>non-monotonic logic</u>.

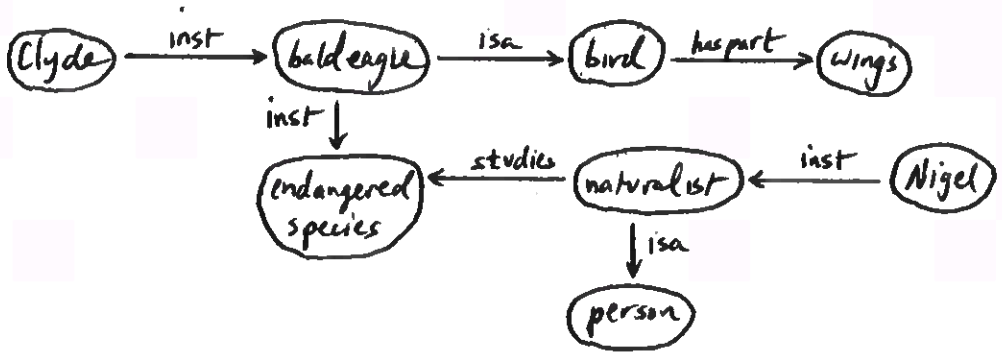
<u>Lack of a clear semantics</u> : <u>isa links</u>

We have not been clear enough about the difference between objects and sets of objects, and between subset and membership. Consider:



We can conclude that Nigel studies Clyde, which may or may not be true.

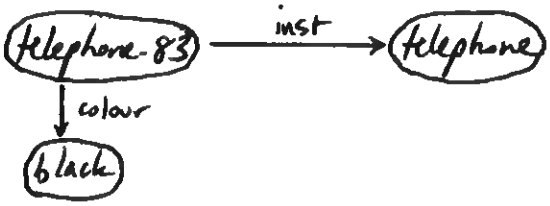
The problem is that some nodes represent individuals and some represent classes of individuals, and isa has been used to represent both membership in a class and inclusion between classes. Some class properties are not inherited by class instances. We can handle this (more or less) by making a distinction between <u>isa</u> for subset and <u>inst</u> (instance of) for membership with different inferences for these different links.



Note that endangered species is really a class of classes, so bald eagle is an instance of endangered species. We might need different kinds of studies links to make a distinction between studying an individual, studying a class of individuals and (the case used here) studying a class of classes of individuals.

Lack of a clear semantics: general problems

In general, what does the notation mean? Consider:



What is telephone-83 supposed to represent in such a network, exactly? The concept of a black telephone? A particular black telephone? An indefinite single black telephone? A typical black telephone? Some kind of assertion about the relationship between telephones and blackness — that telephones are black (presumably just some telephones)? What about telephone: is this the class of all telephones, the concept of a telephone, or a typical telephone?

Different interpretations support different inferences, so these questions are not just hair-splitting.

For more criticism along these lines, see the classic paper "What's in a link: foundations for semantic networks" by W.A. Woods in Brachman + Levesque, Readings in Knowledge Representation.

Conclusion

Semantic networks are a popular representation scheme in AI. The notation seems to capture something important about association in the psychology of memory which has the advantage of being easily implementable using pointers.

But the simple idea of having nodes which represent things in the world and links that represent relationships between things can't be pushed too far.

Frames and scripts

Frames are a scheme for representing knowledge about stereotyped objects and situations. They turn out to be pretty similar to semantic networks in many ways although since they are not drawn as diagrams this similarity is not immediately apparent. They are more flexible than semantic networks as we will see.

Frames were originally proposed by Minsky (1975) as a basis for understanding visual perception, natural-language dialogues, and other complex behaviours.

The classic paper:

M. Minsky, "A framework for representing knowledge" in Brachman + Levesque, Readings in Knowledge Representation

Scripts are framelike structures adapted to the representation of stereotyped sequences of events.

Scripts were developed by Schank and Abelson. The ideas are described in a book:

R. Schank and R. Abelson, Scripts, plans, goals and understanding. Lawrence Erlbaum Publishing Co., 1977.

Frames

A frame is basically a record as in Pascal, with a number of fields called *slots*, containing values called *fillers*.

You can think of a frame as being a complex node in a semantic network, with one slot being filled with the name of the object that the frame represents and the other slots being filled with the values of various attributes associated with the object.

Like semantic networks, frames make use of defaults and inheritance of properties between frames which are related by an *isa* (or *a kind of*) relation. The set of frames in a system form a hierarchy with higher levels in the system defining slots which lower level frames should provide with fillers, sometimes with a range of possible values or a default value. It is possible to provide a procedure that can be called to provide fillers rather than just a value (the posh term for this is *procedural attachment*).

This organization is supposed to facilitate expectation-driven processing, in which one looks for things that are expected in the context one thinks one is in — i.e., one looks for fillers to put into slots in the current frame.

For example (note that the notation is not standardized!):

Person : a kind of *Thing* with
- *Age*
- *Height*
- *Weight*

Property : a kind of Thing with
    Units
    Range
Age : a Property with
    Units : Years
    Range : 0-120
Height : a Property with
    Units : Feet
    Range : 0-8
Weight : a Property with
    Units : Pounds
    Range : 0-300
Fred : a Person with
    Age : 20
    Height : 6
    Weight : 160

The Person frame can be viewed as a declaration — it says what attributes a person has (age, height, weight). The attributes themselves are represented as frames, which can have properties of their own.

"A kind of" relates classes (called kinds) to classes, i.e. it corresponds to the subset relation. "A" (as in Height : a Property) relates objects (called instances) to classes, i.e. it corresponds to membership.

Another example:

Car : a kind of Thing with
    Country of Manufacture
    Miles per Gallon
    Reliability
Country of Manufacture : a Property with
    Range : (Britain, France, Germany, Italy, Japan)
Miles per Gallon : a Property with
    Range : 0-100
Reliability : a Property with
    Range : (Low, Medium, High)
Citroen : a kind of Car with
    Country of Manufacture : France
    Reliability : Medium
Citroen GS : a kind of Citroen with
    Miles per Gallon : 32
GSX 638 T : a Citroen GS with
    Reliability : Low

The typical car is too heterogeneous to have any of its properties already filled in. However, the typical Citroen is a medium reliable car made in France. Its petrol consumption depends on the model; a Citroen GS has 32 mpg. But my Citroen GS isn't as reliable as a typical Citroen. Thus the frame for my car (GSX638T) inherits its country of manufacture from the Citroen frame, as the Citroen GS frame does, but it overwrites the Reliability slot filler inherited from the Citroen frame. It inherits its miles per gallon from the Citroen GS frame, although this could have been overwritten as well. Thus it remains an instance of its class, even though it violates some of the typical property values in the class structure.

As with semantic nets, the subclass relation is not exactly the same as the subset relation of set theory (since some of the superclass properties may be violated by the subclass), nor is the instance relation the same as set membership. What we have here is more a question of capturing the relationship between generic and individual concepts.

Recall that logic failed to handle defaults which could be overridden (the example of birds which can fly unless they are penguins led to inconsistency). Frames can handle such things as the above example demonstrates. Minsky's paper suggests that logic is inadequate for knowledge representation just because it is too rigid to handle the kind of informal reasoning which takes place in peoples' minds.

The "frame problem"

The "frame problem" discussed by McCarthy and Hayes (in "Some philosophical problems from the standpoint of artificial intelligence", 1969) has to do with the inability of most knowledge representation formalisms to model model side effects of actions — knowing which things in the world are no longer true as the result of an action, and which things are still true (the latter is harder).

The frame problem has <u>nothing to do with frames</u>! Many people have been confused by this terminology. (There is a slight relation insofar as both have to do with modelling expectations, etc. but the same can be said of most concepts in AI.)

Here is a more complicated example, in a different notation:

```
Generic RESTAURANT Frame
        Specialization-of:  Business-Establishment
        Types:
                range:      (Cafeteria, Seat-Yourself, Wait-To-Be-Seated)
                default:    Wait-to-be-Seated
                if-needed:  IF plastic-orange-counter THEN Fast-Food,
                            IF stack-of-trays THEN Cafeteria,
                            IF wait-for-waitress-sign or reservations-made
                                        THEN Wait-To-Be-Seated,
                            OTHERWISE Seat-Yourself.
        Location:
                range:      an ADDRESS
                if-needed:  (Look at the MENU)
        Name:
                if-needed:  (Look at the MENU)
        Food-Style:
                range:      (Burgers, Chinese, American, Seafood, French)
                default:    American
                if-added:   (Update Alternatives of Restaurant)
        Times-of-Operation:
                range:      a Time-of-Day
                default:    open evenings except Mondays
        Payment-Form:
                range:      (Cash, CreditCard, Check, Washing-Dishes-Script)
        Event-Sequence:
                default:    Eat-at-Restaurant Script
        Alternatives:
                range:      all restaurants with same FoodStyle
                if-needed:  (Find all Restaurants with the same FoodStyle)
```

In this example, the Specialization-of slot establishes an inheritance hierarchy, just like "a kind of" in the previous examples. The If-needed subslots contain attached procedures which can be used to fill in the slot's value if necessary. Default suggests a filler for the slot unless there is evidence to the contrary.

Event-Sequence is a script — see below. In this example, the notation and the details are not important; it is included as an example of a non-trivial frame.

Scripts

Scripts are frame-like structures adapted to the representation of stereotyped sequences of events, for example going to a restaurant.

There are different scripts for going to a restaurant, for example the cafeteria script (you choose your food from a display and put it on a tray, then pay before eating) is different from the fast food script (again with trays and paying before eating but with a different procedure for getting food) which is different from a fancy restaurant script (sit down, waitress/waiter takes order, food comes, pay after eating, etc.).

There are various interesting questions about how you decide that a particular script is appropriate in a given situation and about learning new scripts. See Schank + Colby for all the details.

Here is the fancy restaurant script; again, the notation and the details are not important.

```
EAT-AT-RESTAURANT Script

    Props:                  (Restaurant, Money, Food, Menu, Tables, Chairs)
    Roles:                  (Hungry-Persons, Wait-Persons, Chef-Persons)
    Point-of-View:          Hungry-Persons
    Time-of-Occurrence:     (Times-of-Operation of Restaurant)
    Place-of-Occurrence:    (Location of Restaurant)
    Event-Sequence:
        first:    Enter-Restaurant Script
        then:     if (Wait-To-Be-Seated-Sign or Reservations)
                          then Get-Maitre-d's-Attention Script
        then:     Please-Be-Seated Script
        then:     Order-Food-Script
        then:     Eat-Food-Script unless (Long-Wait) when
                          Exit-Restaurant-Angry Script
        then:     if (Food-Quality was better than Palatable)
                          then Compliments-To-The-Chef Script
        then:     Pay-For-It-Script
        finally:  Leave-Restaurant Script
```

Production Systems

Production systems were first discussed by Post in the 1940's and Markov in the 1950's in the context of automata theory.

The basic idea is that of rules consisting of condition - action pairs. An action can be performed provided that its associated enabling condition is satisfied.

These rules are called production rules or just productions.

A production system has 3 parts:

- a rule base = a set of production rules
- a data structure called the context or working memory
- an interpreter

A production rule is a statement of the form: "if this condition holds then this action is appropriate"

For example, the rule

Always drive with headlights after dark.

might be encoded as the production rule

If after-dark and driving then use-headlights

The if part of the rule, called the condition, states the condition(s) that must hold for the production rule to be applicable.

The then part, called the action, gives the appropriate action to take.

During the execution of the system, a rule whose condition is satisfied can fire provided the interpreter selects it from amongst the others which have their conditions satisfied at the same time.

The conditions refer to the context, which can be just a list of symbols or something with more structure. The condition can check whether certain things are present in the context, and the action can then change the context so that other rules will have their conditions satisfied.

The interpreter's main job is deciding which rule to fire when (as is very often the case) there is more than one rule whose condition is satisfied.

Example (from the Handbook of AI)

Here is a very simple production system which can be used to identify food items.

Productions

1. If in-context "green" then add-context "produce".
2. If in-context "packed in small container" then add-context "delicacy".
3. If in-context "refrigerated" or in-context "produce"
   then add-context "perishable".
4. If in-context "heavy" and in-context "inexpensive"
   and not in-context "perishable" then add-context "staple".
5. If in-context "perishable" and in-context "heavy"
   then add-context "turkey".
6. If in-context "heavy" and in-context "produce"
   then add-context "watermelon".

The context here is just a list of symbols. In-context is a predicate which checks if a given symbol is in the list, and add-context adds a symbol to the front of the list.

Interpreter

1. Find all productions whose conditions are satisfied and make them applicable
2. If more than one production is applicable, then deactivate any production whose action adds a duplicate symbol to the context.
3. Execute the action of the lowest-numbered applicable production. If no production is applicable then quit.
4. Reset the applicability of all productions and return to 1.

The interpreter operates in a cycle. In each cycle, the productions are examined to see which are appropriate for firing. If more than one is appropriate, a single production is selected from among them. Finally, that production is fired. These three phases are called matching, conflict resolution and action.

In our example, the conflict resolution phase first filters out rules which would not produce any interesting change to the context (this prevents cycles as long as rules are only allowed to add to the context) and then selects the first of all those which are applicable. Typically, conflict resolution strategies are considerably more complicated.

Here is what happens if we run this production system starting from the initial context [green, heavy].

context = [green, heavy]
(only rule 1 is applicable)
⇒ context = [produce, green, heavy]
(rules 1, 3, 6 are applicable; rule 1 adds a duplicate; we select rule 3)
⇒ context = [perishable, produce, green, heavy]
(rules 1, 3, 5, 6 are applicable; rules 1, 3 add duplicates; we select rule 5)
⇒ context = [turkey, perishable, produce, green, heavy]
(rules 1, 3, 5, 6 are applicable; rules 1, 3, 5 add duplicates so we apply rule 6)
⇒ context = [watermelon, turkey, perishable, produce, green, heavy]
(rules 1, 3, 5, 6 are applicable; all add duplicates so we quit.)

If we adopt the convention that the answer is the first symbol in the context then the result is <u>watermelon</u>.

Rule 5 should be fixed somehow to ensure that it does <u>not</u> fire in this example — we probably want to add the condition that "green" or maybe "produce" is not in the context.

One of the advantages of production systems is supposed to be that bugs like this are easy to patch up — but on the other hand this kind of testing and patching is not the way to proceed if you want a system which works properly when presented with a previously untried input.

Some issues in the design of production systems

- Complexity of conditions and actions

To make rules more general, we can allow them to contain variables which are bound during matching and whose scope extends to the end of the rule. Compare (using a Prolog-like syntax):

> if age(Peter, 36) and unemployed(Peter)
> then claim(Peter, unemployment-benefit)

which only applies to Peter when he is 36, and:

> if age(P, A) and unemployed(P)
> and $15 \leq A$ and $A \leq 65$
> then claim(P, unemployment-benefit)

Often the condition and action can be arbitrary hunks of LISP code. This gives great power, but there is also great potential for making an enormous mess of things if the power is used in an undisciplined way.

- Structure of the rule base and context

It is desirable to structure the rule base if it is big so as to allow rapid determination of the rules which are applicable situation. This can be a matter of "compiling" the rules into a decision tree or partitioning them according to their conditions so that only a relatively small number of rules have to be examined during the matching phase (cf. Rete algorithm).

The context can also be structured to allow it to represent more complicated situations. For example, some systems use a semantic network as context.

- Conflict resolution

There are many strategies for choosing between all the rules which are applicable on any given cycle. Some of the different approaches:

Choose:

1. The first rule (according to the order of rules in the rule base)
2. The highest priority rule
3. The most specific rule (the one with the most detailed condition which is satisfied)
4. The rule that refers to the most recently added element of the context.
5. A new rule (one which has not been used previously — or, avoid the most recently used rule)
6. An arbitrary rule
7. Don't choose — explore all choices in parallel

Terminology:

Refractoriness: Don't use any rule more than once on the same data (see 5 above)

Recency: Use most recently added data in preference to data which has been around for awhile (4 above)

Specificity: Rules having conditions which are more difficult to satisfy are preferred (3 above)

- Metarules

The production system may include rules which control the application of other rules, e.g. in a medical diagnosis system like MYCIN we might have a rule which says that if we are dealing with an alcoholic or a burn patient then consider rules which mention certain diseases before rules which mention other diseases.

See "A short introduction to OPS5" by Peter Ross for information about a typical production rule language.

Rete matching

Pattern matching in production systems

Recall that the first phase of the recognize-act cycle is *matching*, where all the conditions of all the production rules are checked against the context to see which ones are satisfied. This gives rise to the *conflict set*.

In a production rule language like OPS5 (let us adopt this language for the rest of this lecture) the conditions can include *variables* which are bound during matching. That is, matching is a kind of *pattern matching*. The example in the last lecture (with the watermelon) didn't show this — there all the conditions were just searches through the context for particular items.

In the presence of variables, matching involves finding instantiations for the variables such that the conditions are satisfied. There might be several ways of instantiating the variables so as to satisfy a given condition. The conflict set is not just the *rules* which can fire, but also the *instantiations* which allow them to fire. Thus the conflict set can be bigger than the set of rules.

Matching is by far the most expensive thing which a production system interpreter has to do. It has to find *all* the instantiations of *every* rule which makes it succeed — this potentially involves matching the condition of every rule against every element in the context on every cycle.

Since production rule systems can contain several thousand rules, and the number of elements in the context can be large, this is a problem. If there are p productions and c context elements then the run time of the obvious algorithm is proportional to pc. Even if there is some clever way of finding matching context elements for a given rule without searching through the whole context (maybe some kind of hashing scheme), run time is still proportional to p. One place that work is being wasted is that often in successive cycles the same context elements will be matched against the same conditions giving the same result since these elements were not changed by the last action.

The Rete match algorithm

Reference: C.L. Forgy, "Rete: a fast algorithm for the many pattern / many object pattern match problem" Artificial Intelligence 19 (1982).

This is used as the basic pattern matching mechanism in OPS5 and other production rule interpreters in the OPS family.

The algorithm is based on two simple observations:

- The conditions of different productions often share common subconditions. A naive matching algorithm will match each of these subconditions against the context each time it occurs.
- The context is only modified a little bit on each cycle, so most of the new conflict set will be exactly the same as the old conflict set. Naive approaches don't take advantage of this.

The algorithm tackles the first source of inefficiency by compiling the productions into a tree-structured network where common subexpressions only appear once. The second source of inefficiency is taken care of by using the change in the context as input to the matcher rather than the new context itself. Thus the old conflict set can be updated by removing matches which no longer work (since some context element has been removed or changed) and adding new matches.

The algorithm thus takes as input a set of tokens describing changes to the context (added or deleted elements) and produces as output a set of changes to the conflict set. A modification of an element in the context (as opposed to an addition or deletion) is represented by a deletion followed by an addition, to make things simpler.

OPS5 context elements are of the form

(class-of-element ↑attribute$_1$ value$_1$ ... ↑attribute$_n$ value$_n$)

A token is thus of the form

<+ (class-of-element ↑attribute$_1$ value$_1$ ...)>

or <- (class-of-element ↑attribute$_1$ value$_1$ ...)>

The network

Consider a pattern:

(Expression ↑Name <N> ↑Arg1 0 ↑Op + ↑Arg2 <X>)

Matching this against the context means finding an element having the following features:

- the class must be Expression
- the value of the Arg1 attribute must be 0
- the value of the Op attribute must be +

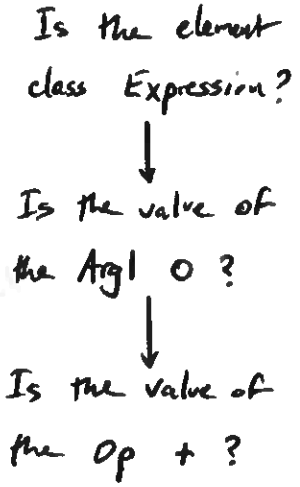
Consider a rule which contains several such patterns:

```
(P PlusOx
    (Goal ↑Type Simplify ↑Object <N>)
    (Expression ↑Name <N> ↑Arg1 0 ↑Op + ↑Arg2 <X>)
--> ... )
```

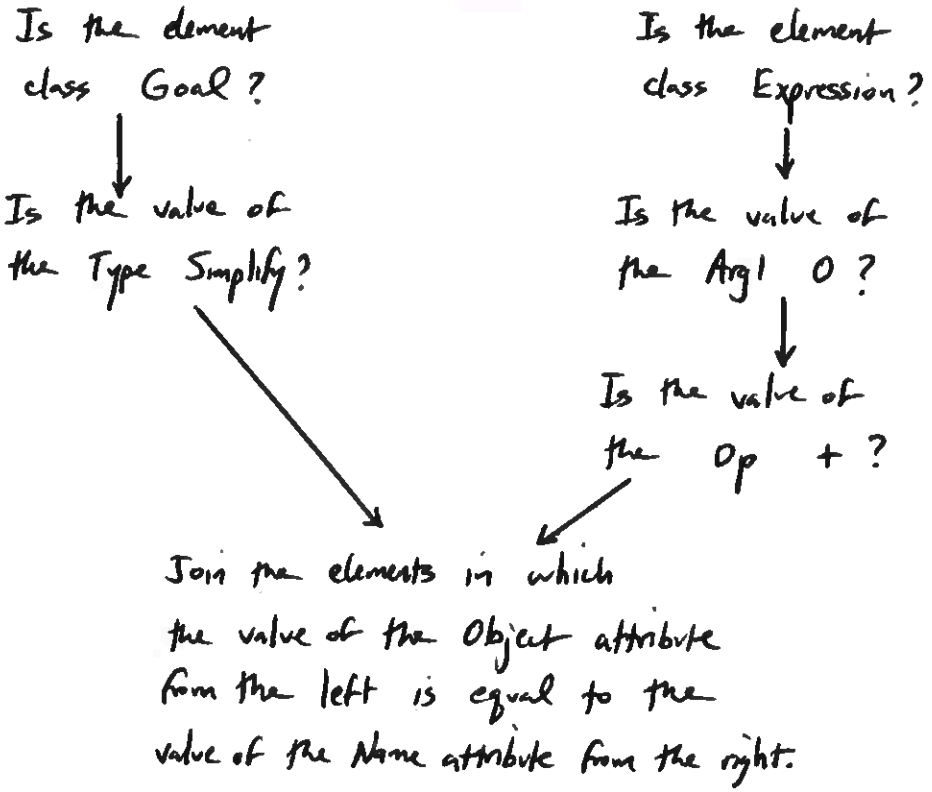
In addition to finding one element which has the features mentioned before, we now have to find another element having the features required by the first pattern, and we <u>also</u> have to check that the Object attribute of one is the same as the Name attribute of the other.

The algorithm builds a network by linking together nodes which test for features. Token are fed through the network which reports whether or not the given elements (the ones which the tokens refer to) are matched by the left-hand side of any production.

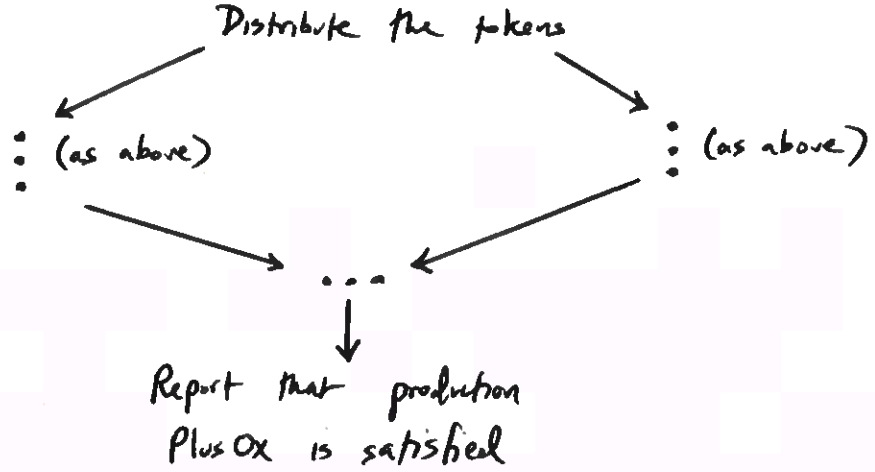
Checking for an element matching a single pattern gives rise to a linear sequence of nodes, e.g.



Such linear sequences are then put together to find the pairs of elements having the required matching bits. For example (see the rule PlusOx above):



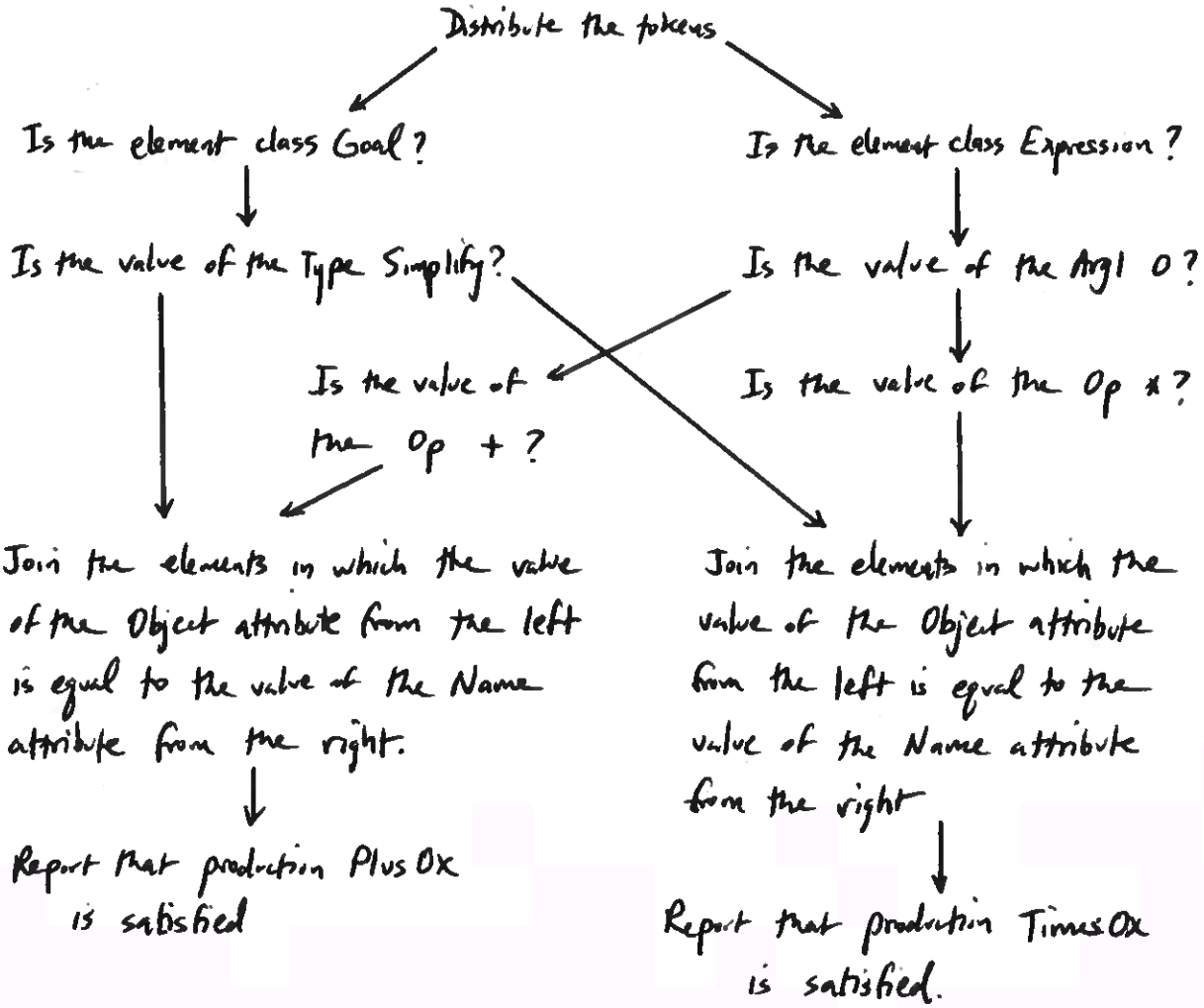
There is also a node at the top which distributes the tokens to all the bits of the network and special nodes for each production at the bottom that report that the production is satisfied:



Often there will be other productions which share common subexpressions. For example:

```
(P PlusOx
   (Goal ↑Type Simplify ↑Object <N>)
   (Expression ↑Name <N> ↑Arg1 0 ↑Op + ↑Arg2 <X>)
-->...)
(P TimesOx
   (Goal ↑Type Simplify ↑Object <N>)
   (Expression ↑Name <N> ↑Arg1 0 ↑Op * ↑Arg2 <X>)
-->...)
```

The network compiler builds a network which shares the appropriate bits.



Consider what happens when the following two elements are added to an empty context:

(Goal ↑Type Simplify ↑Object Expr17)
(Expression ↑Name Expr17 ↑Arg1 0 ↑Op * ↑Arg2 A)

First the token <+ (Goal ↑Type Simplify ↑Object Expr17)> is created and sent to the root of the network, which sends the token to its two successors.

The node on the right rejects the token because its class is not Expression. The one on the left accepts it and passes it on to its successor, which accepts it and passes it on to both of its successors. Since no token has arrived on the other branch, the token is stored for the moment.

When the token <+ (Expression ↑Name Expr17 ↑Arg1 0 ↑Op * ↑Arg2 A)> is processed, it is rejected by the left-hand bit of the network and accepted by the right-hand node and its successor. It is accepted by the node which is looking for Op * but not by the one which is looking for Op +. The join node on the right now has two tokens, one from each branch, so it can go to work checking that the common bits of the two tokens match. It then sends out the token

<+ (Goal ↑Type Simplify ↑Object Expr17)
(Expression ↑Name Expr17 ↑Arg1 0 ↑Op * ↑Arg2 A)>

When its successor, the terminal node for TimeOx, receives this token, it adds (because the token contains a +) the instantiation of TimeOx to the conflict set.

You can see from this example that the two-input nodes have to maintain queues of inputs, one for each incoming arrow. The + and − tokens are processed identically except:

- the terminal node uses the + or − to determine whether to add or remove an instantiation from the conflict set.
- the two-input nodes delete any matching + tokens from their input queues when a − token is received.

We need a different kind of node to handle negated patterns — see the reference for details.

Some Pros and Cons of Production Systems

Production systems are used in AI for building expert systems. They are used to represent a body of knowledge concerning how people do a specific real-world task (medical diagnosis, mineral exploration, etc.). The standard example is MYCIN. Production systems provide a nice mix between declarative and procedural knowledge which is appropriate for this kind of job.

Production systems are also used in psychology to model human behaviour. They are attractive because of their stimulus-response character, which fits well with psychological theories about behaviour.

Advantages of production systems for knowledge representation

Modularity — a production system is just a set of independent rules

- Individual productions in the rule base can be added, deleted or modified independently.
- Productions behave much like independent pieces of knowledge
- Changing one rule can be accomplished without having to worry about direct effects on the other rules, since rules only communicate via the context — i.e. they can't call one another directly.
- This comparative modularity is important in building large rule bases — we know what a given rule will mean in whatever rule base it is used, independent of the situation
- But modularity is hard to maintain in large systems.

Uniformity

- A uniform structure is imposed on all knowledge in the rule base
- This means (supposedly) that it can be more easily understood by another person or by the system itself, at least compared with e.g. semantic net representations.
  An example is Waterman's poker-playing program which started with a simple set of productions which it extended and improved as it gained experience in playing the game.
  Reference: D.A. Waterman, "Generalization learning techniques for automating the learning of heuristics", Artificial Intelligence 1 (1970).
  On the other hand, Winston's concept-learning program does a similar thing with semantic nets.

Naturalness

- It is easy to represent certain important kinds of knowledge
- Especially, rules about what to do in predetermined situations can be naturally encoded into production rules.
- These are the kinds of statements which are most often used by human experts to explain how they work.
  But this only works up to a point — a good expert has to understand why, which is just what a production rule does not encode.

Disadvantages of production systems

Inefficiency

- Modularity and uniformity of productions result in a high overhead
- Since production systems perform every action by means of the match-action cycle and convey all information by means of the context, it is difficult to make them efficiently responsive to predetermined sequences of situations (or to take larger steps in their reasoning when the situation demands it).
- There is a tradeoff between rules which are natural and rules which are efficient.

Opacity

- It is hard to follow the flow of control in problem solving
- Algorithms are not as clear as they would be if expressed in a programming language. Although situation-action knowledge can be expressed naturally in production systems, algorithmic knowledge is not expressed naturally.
- Productions are isolated (they don't call each other) and they are of uniform "size" (there is nothing like a subroutine hierarchy in which one production can be composed of several subproductions)
- Function calls and subroutines (added somehow) would help to make the flow of control easier to follow.
- Loops and recursion (or nested control or data structures of any kind) are not handled naturally.

Other disadvantages

- Productions are well-suited to encoding empirical knowledge, but they are very much less effective for encoding more subtle knowledge about causality etc. An example of this was the Guidon system, which attempted to use MYCIN's rule base for teaching — the problem was that the "why" component is absent.
- The uniform syntax of production rules hides the fact that such rules often perform very different functions
- Certain structural and strategic decisions about the representation of domain knowledge are implicit in the rules.
- Things like the ordering of conditions and the order of rules in the rule base are vital to the way that a production system performs. Such things are a crucial aspect of the expert knowledge that is encoded (try this before this) but this aspect is not represented explicitly. Ordering can be either vitally important or it can be completely unimportant (in some cases it doesn't matter at all) and there is nothing which distinguishes between these two situations.

Worth reading:
W.J. Clancey, "The epistemology of a rule-based expert system: a framework for explanation", Artificial Intelligence 20 (1983)

One way to get the advantages of production systems and avoid the disadvantages is to mix production rules with other knowledge representation formalisms.

Production rules (e.g. OPS5) as a programming language

OPS5 is easy to learn, and it is relatively easy to write simple programs in OPS5. But a little experience with OPS5 reveals some problems.

There is nothing in the syntax which helps the programmer make and maintain certain semantic distinctions, such as that between essential, static properties of an object that cannot change during a program run, and accidental, dynamic properties that do change. Making such distinctions explicit can help program design, debugging and modification, especially in large systems that will be worked on by many people. On the other hand, there is some overhead in terms of declarations and restrictions which have to be specified (this is the strongly typed vs. untyped controversy).

There is a lot more to writing a good rule-based program than just formulating rules which are true according to an expert and encoding them as production rules. Even if the rules do encode true generalizations about the domain, there is no guarantee that the program will perform as expected. It is necessary to write the rules carefully with the conflict resolution strategy in mind. It is difficult to predict the outcome of competition between all the rules which are vying for attention on each cycle.

This means that earlier claims about modularity etc. have to be taken with a grain of salt. In complicated situations (because of conflicts) it is almost a matter of chance what happens. By removing or changing a rule which was used at a critical time the behaviour of the system can change in strange ways.

Representing knowledge as an unordered and unstructured set of rules is probably a bad idea. It fails to take advantage of whatever explicit structure the domain possesses in terms of taxonomic, part-whole or cause-effect relations between objects and classes of objects. It also encourages undisciplined systems which only work by accident in certain cases (useful for demonstrations but not for much else).

Backtracking is ruled out since modifications to the context are destructive, making it difficult to return to an earlier state of the computation, at least in most systems. This is efficient and simple, but restrictive in comparison with e.g. Prolog, and there are times when this can cause problems.

Appropriate domains for production systems

The pluses and minuses mentioned suggest that production systems are good for certain purposes but not for others. In "An overview of production systems" (*Machine Intelligence* 8, E. Elcock and D. Michie, eds.) by R. Davis and J.J. King (1977) the following points are given as features of appropriate domains:

- Domains in which the knowledge is diffuse, consisting of many facts (e.g. clinical medicine) as opposed to domains in which there is a concise unified theory (e.g. physics)
- Domains in which processes can be expressed as a set of independent actions (a medical patient-monitoring system) as opposed to domains with dependent subprocesses (a payroll program)
- Domains in which knowledge can be easily separated from the manner in which it is to be used, as opposed to cases in which representation and control are merged (as in a recipe)

This was rephrased in AI terms in "Production systems as a programming language for artificial intelligence applications" (CMU research report, 1976) by M.D. Rychener:
If we can view the task as a sequence of transitions from one state to another in a problem space, we can model this behaviour with production systems, since each transition can be effectively represented by one or more production firings.

Production systems offer great flexibility, but not much in the way of high-level facilities. They are most useful when the kind of knowledge to be captured does not have a small, clean conceptual basis, and when the rule set may be large but there is no great interdependence between the rules.

The tradeoff between expressiveness and tractability

Introduction

The role of a knowledge representation (KR) system is to perform a class of inferences determined by the propositions in the knowledge base (KB). One thing it should be able to do is determine when the truth of one proposition is implicit in another. But depending on the range of propositions which have to be dealt with, this task can be either relatively trivial or completely unsolvable.

As long as we are dealing with computational systems that reason automatically and correctly, they will either be limited in what knowledge they can represent or unlimited in the reasoning effort they might require.

Reference:
Levesque and Brachman, "A fundamental tradeoff in knowledge representation and reasoning" in *Readings in Knowledge Representation.*

Knowledge Representation

The job of a KR system is to manage a KB for a knowledge based system. The main problem is determining what the KB says regarding the truth of certain propositions. It is not whether the proposition itself is present in the KB that counts, but whether its truth is implicit in the KB. That is, a KR has to be able to determine, given a proposition S, the answer to the question:

> Assuming the world is such that what is believed is true, is S also true?

Determining the answer to this question might require not just simple retrieval capabilities, but also *inference* of some sort.

First-order logic

Deciding whether or not a sentence of first-order logic (FOL) is a theorem is unsolvable (i.e. the set of theorems is recursively enumerable). This is bad if we want to use the KR system for representation of routine common sense knowledge: bogging down on a logically difficult but low-level subgoal and being unable to continue without human intervention is clearly an unreasonable form of behaviour for an "intelligent" system. If the KR service is going to be used as a utility and is not available for inspection or control then it should be dependable both in terms of correctness and the resources it consumes. Of course, the same problem arises for any language which is at least as powerful as FOL, for example English.

There are two ways to avoid the problem. Speeding up the theorem prover helps but does not solve the problem. Another way is to relax our notion of correctness, e.g. by making the theorem prover always return an answer after a certain amount of time ("unknown" if it hasn't been able to complete a proof one way or the other). This might be acceptable under some circumstances but it is not a solution either.

The power of FOL is used in KR not to deal with infinite domains (this is its main use in mathematics) but to deal with *incomplete knowledge*. For example:

1. $\neg$student(john)
   This says that John is not a student without saying what he is.
2. parent(sue,bill) $\vee$ parent(sue,george)
   This says that either Bill or George is a parent of Sue, but does not specify which.
3. $\exists x.$ (cousin(bill,x) $\wedge$ male(x))
   This says that Bill has at least one male cousin but does not say who.
4. $\forall x.$ (friend(george,x) $\Rightarrow \exists y.$ child(x,y))
   This says that all of George's friends have children without saying who those friends or their children are or even if there are any.

The main point is that FOL is not used to capture complex details about the domain, but to avoid having to represent details that may not be known. The expressive power of FOL determines not so much what can be said, but what can be left unsaid.

Last solution to the tractability problem: restrict the logical form of the KB by controlling the incompleteness of the knowledge represented.

Database form

First type of restriction: *database form*, i.e. restrict KB so that it can only contain the kind of information that can be represented in a standard database. So for example, here is a small database containing information about university courses:

ID	NAME	DEPARTMENT	ENROLLMENT	LECTURER
cs248	Prog. Languages	Computer Science	42	F. Fortran
ma100	Linear Algebra	Mathematics	64	J. Vector
cs373	Anal. of Algorithms	Computer Science	52	H. Polynomial

Translating this into FOL gives something like the following:

course(cs248) dept(cs248,computerscience) enrollment(cs248,42) .....
course(ma100) dept(ma100,mathematics) enrollment(ma100,64) .....

The list of sentences does *not* include ones like:

$$\text{dept(ma100,mathematics)} \vee \text{dept(ma100,history)}$$

so the range of uncertainty we are dealing with is quite limited.

Now, consider the question "How many courses are offered by the Computer Science department?". The knowledge expressed in the above FOL sentences is insufficient to answer this question since there is nothing saying that cs248 and cs373 are different courses, and nothing saying that there are not other courses offered by the Computer Science department which are not listed in the database. But from a database point of view we *could* answer this question. This is because there is some additional information *implicit* in the use of database form, namely the assertion that each constant represents a unique individual:

$$\text{cs248} \neq \text{ma100} \quad \text{mathematics} \neq \text{computerscience} \quad \ldots\ldots$$

and that the only objects of interest are the ones explicitly mentioned (the *closed world assumption*):

$$\forall x.\ (\text{course}(x) \Rightarrow x{=}\text{cs248} \vee x{=}\text{ma100} \vee x{=}\text{cs373})$$

With this additional information we can answer the question.

An important property of a KB in database form is that (except for the implicit information) we are restricted to a subset of FOL which does not include negation, disjunction or existential quantification. This means that it is impossible to express incomplete knowledge, and so inference can be reduced to calculation (counting). We do not have to reason by cases or by contradiction, for instance. If we knew that either cs148 or cs149 or both were Computer Science courses but that no Computer Science course other than cs373 had an odd identification number, we could still determine that there were three CS courses, but not simply by counting. But a KB in database form does not allow us to express this kind of uncertainty, which makes things more tractable.

In a sense, the database is an *analogue* or *model* of the domain, in the sense that there is a very close structural correspondence between the domain and the representation (exactly one representational object for each object in the domain). The advantage of this is that calculation on the model can play the role of more general reasoning techniques, just as arithmetic can replace reasoning with axioms in FOL defining addition and multiplication. The disadvantage is that it is not possible to leave anything unsaid about the domain. The same is true for other kinds of models. For example, a map cannot express the information that a river passes through one of two widely separated towns without specifying which. A plastic model of a ship cannot tell us that the ship it represents

does not have two smokestacks without also telling us how many it does have. This is not to say that there is no uncertainty associated with the analogue, but that this uncertainty is due to the coarseness of the analogue (e.g. how carefully the maps is drawn) rather than to its content.

Logic program form

Second type of restriction: *logic program form:* restrict KB so that it can only contain *Horn clauses*, i.e. FOL sentences of the form:

$$\forall x_1 \ldots x_n . (P_1 \wedge \ldots \wedge P_m \Rightarrow P_{m+1}) \qquad \text{where } m \geq 0 \text{ and each } P_i \text{ is atomic}$$

In the case where m=0 and n=0, the logic program form coincides with the database form.

Again there is information implicit in a logic program which is not accounted for by such a list of sentences, namely a list of sentences (usually infinite) of the form $s \neq t$ for all distinct ground terms s and t, together with a version of the closed world assumption which is now a set containing the negation of every ground atomic sentence not implied by the Horn clauses in the explicit KB.

The result of this restriction is a KB which once again has complete knowledge of the world, but this time may require inference to answer questions. The reasoning in this case is the *execution* of the logic program. For example, given a Prolog KB consisting of:

```
parent(bill,mary).
parent(bill,sam).
mother(X,Y) :- parent(X,Y), female(Y).
female(mary).
```

we know exactly who the mother of Bill is, but only after having executed the program.

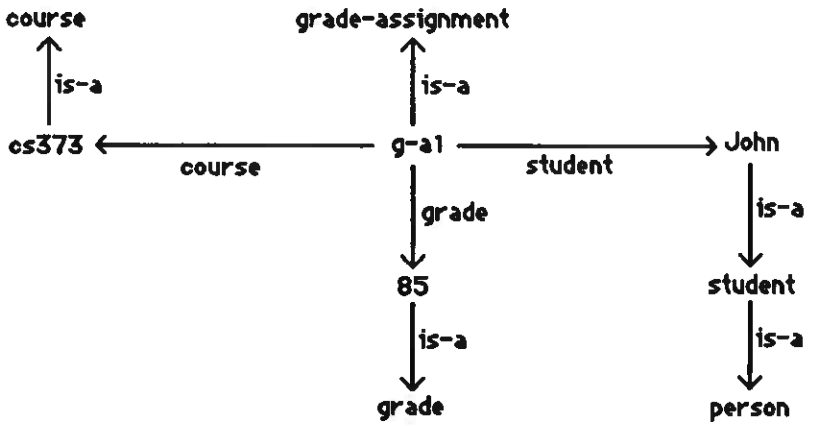
In a sense, the logic program form does not provide any computational advantage to a KB in unrestricted FOL since determining what is implicit in the KB is in general undecidable. On the other hand, the form is much more manageable than in the general case since the necessary inference can be split very nicely into two components: a *retrieval* component that extracts (atomic) facts from a database by pattern-matching and a *search* component that tries to use the non-atomic Horn clauses to complete the inference.

Semantic network form

Third type of restriction: *semantic network form.* This amounts to an FOL in database form containing only unary and binary predicates. For example, instead of representing the fact that John's grade in cs373 was 85 by

grade(john,cs373,85)

we have to postulate the existance of objects called "grade-assignments" and represent the fact about John in terms of a particular grade-assignment g-a1 as:



which amounts to the following:

grade-assignment(g-a1) ∧ student(g-a1,john) ∧ course(g-a1,cs373)
∧ grade(g-a1,85)

The main feature of a semantic net is not the treatment of individuals but the treatment of the unary predicates (*types*) and binary predicates (*attributes*). The types are organized into a *taxonomy* which we will represent as a set of sentences of the form

$\forall x.\ (B(x) \Rightarrow A(x))$

(This is not a very good example because attributes are only attached to constants, not to types, and the taxonomy is very simple.) The nodes are either constants or types, and the edges are either labelled with an attribute or with the special label *is-a*. (Note that the interpretation of an edge depends on whether its source and target are constants or types. For example, from a constant c to a type B, is-a says B(c), but from a type B to a type A, is-a gives rise to a taxonomic sentence $\forall x.\ (B(x) \Rightarrow A(x))$.)

The significance of this graphical representation is that it allows certain kinds of inferences to be performed by simple graph-searching techniques. For example, to kind out if a particular individual has a certain attribute, it is sufficient to search from the constant representing that individual, up is-a links, for a node having an edge labelled with that attribute. By placing the attribute as high as possible in the taxonomy, all individuals below it can *inherit* the property.

The graph representation suggests different kinds of inference that are based more directly on the structure of the KB than to its logical content. For example, we can ask how two nodes are related and answer by finding a path in the graph between them (*intersection* search). Given for instance Clyde the elephant and Jimmy Carter, we could end up with an answer saying that Clyde is an elephant and the favourite food of elephants is peanuts which is also the major product of the farm owned by Jimmy Carter. A typical method of producing this answer would be to perform a "spreading activation" search beginning at the nodes for Clyde and Jimmy. Obviously, this form of question would be very difficult to answer for a KB that was not in semantic net form.

The appeal of the graphical nature of semantic nets has lead to forms of reasoning, such as default reasoning, that do not fall into standard logical categories and are not yet very well understood. This is a case of a representational notation taking on a life of its own and motivating a completely different style of use not necessarily grounded in truth theory. It is easier to develop an algorithm that appears to reason over structures of a certain kind than to *justify* its reasoning by explaining what the structures are saying about the world.

This is not to say that defaults are not a crucial part of our information about the world. The problem is to make this intuition precise. Paradoxically, the best formal accounts of defaults would claim that reasoning with them is *more* difficult than reasoning without them.

Conclusion

It is more difficult to reason with one representational language than with another and this difficulty increases as the expressive power of the language increases. Expressiveness often amounts to the ability to leave certain things unsaid, with full first-order logic at one extreme and databases at the other. There is a tradeoff between the expressiveness of a representational language and its computational tractability. There is no single best language, only more or less interesting positions on the tradeoff. Neither expressibility nor tractability by itself determines the value of a representation language.