# Logical reasoning systems

- Theorem provers and logic programming languages

- Production systems

- Frame systems and semantic networks

- Description logic systems

# Logical reasoning systems

- Theorem provers and logic programming languages – Provers: use resolution to prove sentences in full FOL. Languages: use backward chaining on restricted set of FOL constructs.

- Production systems – based on implications, with consequents interpreted as action (e.g., insertion & deletion in KB). Based on forward chaining + conflict resolution if several possible actions.

- Frame systems and semantic networks – objects as nodes in a graph, nodes organized as taxonomy, links represent binary relations.

- Description logic systems – evolved from semantic nets. Reason with object classes & relations among them.

# Basic tasks

- Add a new fact to KB – TELL

- Given KB and new fact, derive facts implied by conjunction of KB and new fact. In forward chaining: part of TELL

- Decide if query entailed by KB – ASK

- Decide if query explicitly stored in KB – restricted ASK

- Remove sentence from KB: distinguish between correcting false sentence, forgetting useless sentence, or updating KB re. change in the world.

# Indexing, retrieval & unification

- **Implementing sentences & terms:** define syntax and map sentences onto machine representation.

  Compound: has operator & arguments.
  - e.g., $c = P(x) \wedge Q(x)$         $Op[c] = \wedge;$ $Args[c] = [P(x), Q(x)]$

- **FETCH:** find sentences in KB that have same structure as query. ASK makes multiple calls to FETCH.

- **STORE:** add each conjunct of sentence to KB. Used by TELL.
  - e.g., implement KB as list of conjuncts
  - TELL(KB, $A \wedge \neg B$) TELL(KB, $\neg C \wedge D$)
  - then KB contains: $[A, \neg B, \neg C, D]$

## Complexity

- With previous approach,

   FETCH takes O(n) time on n-element KB

   STORE takes O(n) time on n-element KB (if check for duplicates)


Faster solution?

# Table-based indexing

- **What are you indexing on? Predicates (relations/functions).**
  **Example:**

| Key | Positive | Negative | Conclu-sion | Premise |
|-----|----------|----------|-------------|---------|
| Mother | Mother(ann,sam)<br>Mother(grace,joe) | -Mother(ann,al) | xxxx | xxxx |
| dog | dog(rover)<br>dog(fido) | -dog(alice) | xxxx | xxxx |

# Table-based indexing

- Use hash table to avoid looping over entire KB for each TELL or FETCH

  e.g., if only allowed literals are single letters, use a 26-element array to store their values.

- More generally:
  - convert to Horn form
  - index table by predicate symbol
  - for each symbol, store:
    - list of positive literals
    - list of negative literals
    - list of sentences in which predicate is in conclusion
    - list of sentences in which predicate is in premise

# Tree-based indexing

- Hash table impractical if many clauses for a given predicate symbol

- Tree-based indexing (or more generally combined indexing):
  compute indexing key from predicate and argument symbols

Predicate?

First arg?
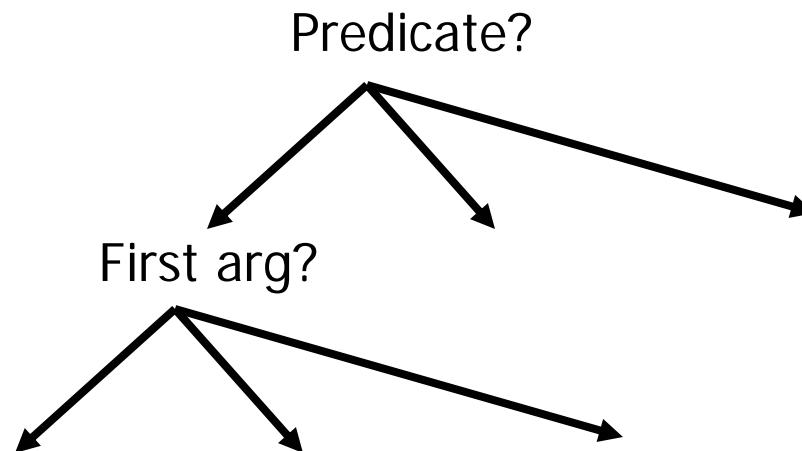
**Tree-based indexing**

**Example:**

Person(age,height,weight,income)
Person(30,72,210,45000)
Fetch( Person(age,72,210,income))
Fetch(Person(age,height>72,weight<210,income))

## Unification algorithm: Example

Understands(mary,x) implies Loves(mary,x)

Understands(mary,pete) allows the system to substitute pete for x and make the implication that IF

Understands(mary,pete) THEN    Loves(mary,pete)

# Unification algorithm

- Using clever indexing, can reduce number of calls to unification

- Still, unification called very often (at basis of modus ponens) => need efficient implementation.

- See AIMA p. 303 for example of algorithm with O(n^2) complexity

  (n being size of expressions being unified).

# Logic programming

Remember: knowledge engineering vs. programming...

Sound bite: computation as inference on logical KBs

| | Logic programming | Ordinary programming |
|---|---|---|
| 1. | Identify problem | Identify problem |
| 2. | Assemble information | Assemble information |
| 3. | Tea break | Figure out solution |
| 4. | Encode information in KB | Program solution |
| 5. | Encode problem instance as facts | Encode problem instance as data |
| 6. | Ask queries | Apply program to data |
| 7. | Find false facts | Debug procedural errors |

Should be easier to debug $Capital(NewYork, US)$ than $x := x + 2$ !

# Logic programming systems

e.g., **Prolog:**

- Program = sequence of sentences (implicitly conjoined)
- All variables implicitly universally quantified
- Variables in different sentences considered distinct
- Horn clause sentences only (= atomic sentences or sentences with no negated antecedent and atomic consequent)
- Terms = constant symbols, variables or functional terms
- Queries = conjunctions, disjunctions, variables, functional terms
- Instead of negated antecedents, use negation as failure operator: goal NOT P considered proved if system fails to prove P
- Syntactically distinct objects refer to distinct objects
- Many built-in predicates (arithmetic, I/O, etc)

# Prolog systems

Basis: backward chaining with Horn clauses + bells & whistles
Widely used in Europe, Japan (basis of 5th Generation project)
Compilation techniques $\Rightarrow$ 10 million LIPS

Program = set of clauses = head :- literal$_1$, ... literal$_n$.
Efficient unification by open coding
Efficient retrieval of matching clauses by direct linking
Depth-first, left-to-right backward chaining
Built-in predicates for arithmetic etc., e.g., X is Y*Z+3
Closed-world assumption ("negation as failure")
    e.g., not PhD(X) succeeds if PhD(X) fails

# Basic syntax of facts, rules and queries

```
<fact>  ::= <term> .
<rule>  ::= <term> :- <term> .
<query> ::= <term> .
<term>  ::= <number> | <atom> | <variable>
            | <atom> (<terms>)
<terms> ::= <term> | <term>, <terms>
```

# A PROLOG Program

- A PROLOG program is a set of **_facts_** and **_rules_**.

- A simple program with just facts :

```
parent(alice,  jim).
parent(jim,    tim).
parent(jim,    dave).
parent(jim,    sharon).
parent(tim,    james).
parent(tim,    thomas).
```

## A PROLOG Program

- c.f. a table in a relational database.

- Each line is a *fact* (a.k.a. a tuple or a row).

- Each line states that some person X is a parent of some (other) person Y.

- In GNU PROLOG the program is kept in an ASCII file.

## A PROLOG Query

- Now we can ask PROLOG questions :

```
| ?- parent(alice, jim).
yes
| ?- parent(jim, herbert).
no
| ?-
```

- Not very exciting. But what about this :

```
| ?- parent(alice, Who).
Who = jim
yes
| ?-
```

- `Who` is called a *logical variable*.
  - PROLOG will set a logical variable to any value which makes the query succeed.

# A PROLOG Query II

- Sometimes there is more than one correct answer to a query.
- PROLOG gives the answers one at a time. To get the next answer type `;`.

```
| ?- parent(jim, Who).
Who = tim ? ;
Who = dave ? ;
Who = sharon ? ;
yes
| ?-
```

NB: The `;` do not actually appear on the screen.

```
| ?- parent(jim, Who).
Who = tim ? ;
Who = dave ? ;
Who = sharon ? ;
yes
| ?-
```

> NB : The ;
>   do not
>   actually
> appear on
> the screen.

- After finding that `jim` was a parent of `sharon` GNU PROLOG detects that there are no more alternatives for `parent` and ends the search.

## Prolog example

Depth-first search from a start state X:

```
dfs(X) :- goal(X).
dfs(X) :- successor(X,S),dfs(S).
```

No need to loop over S: successor succeeds for each

Appending two lists to produce a third:

```
append([],Y,Y).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
```

```
query:    append(A,B,[1,2]) ?
answers:  A=[]     B=[1,2]
          A=[1]    B=[2]
          A=[1,2] B=[]
```

**Append**

- **append([], L, L)**
- **append([H| L1], L2, [H| L3]) :-  append(L1, L2, L3)**

- Example join [a, b, c] with [d, e].
  - [a, b, c] has the recursive structure [a| [b, c] ].
  - Then the rule says:
  - IF [b,c] appends with [d, e] to form [b, c, d, e]  THEN [a|[b, c]] appends with [d,e] to form [a|[b, c, d, e]]
  - i.e. [a, b, c]                              [a, b, c, d, e]

# Expanding Prolog

- Parallelization:

    OR-parallelism: goal may unify with many different literals and
      implications in KB

    AND-parallelism: solve each conjunct in body of an implication
      in parallel

- Compilation: generate built-in theorem prover for different
  predicates in KB

- Optimization: for example through re-ordering
  e.g., "what is the income of the spouse of the president?"

    Income(s, i) $\wedge$ Married(s, p) $\wedge$ Occupation(p, President)

  faster if re-ordered as:

    Occupation(p, President) $\wedge$ Married(s, p) $\wedge$ Income(s, i)

# Theorem provers

- Differ from logic programming languages in that:

    - accept full FOL

    - results independent of form in which KB entered

# OTTER

- Organized Techniques for Theorem Proving and Effective Research (McCune, 1992)

- Set of support (sos): set of clauses defining facts about  problem
- Each resolution step: resolves member of sos against other axiom
- Usable axioms (outside sos): provide background knowledge about domain
- Rewrites (or demodulators): define canonical forms into which terms can be simplified.   E.g., $x+0=x$
- Control strategy: defined by set of parameters and clauses. E.g., heuristic function to control search, filtering function to eliminate uninteresting subgoals.

# OTTER

- Operation: resolve elements of sos against usable axioms

- Use best-first search: heuristic function measures "weight" of each clause (lighter weight preferred; thus in general weight correlated with size/difficulty)

- At each step: move lightest close in sos to usable list, and add to usable list consequences of resolving that close against usable list

- Halt: when refutation found or sos empty

# Example

File  Edit  View  Go  Communicator                                     Help

Bookmarks  Location: http://www-unix.mcs.anl.gov/AR/otter/     What's Related

Google  Gscout  Library  WoS  PubMed  INSPEC  JTrack  ResearchIndex  WebAdmin  CVSweb  HC Params

## Otter: An Automated Deduction System

Updated August 13, 2001.

## Contents

1. Description
2. Computational Environment
3. AvailabilityVersion 3.2 🎇
4. Documentation
5. Example Inputs 🎇
6. Recent Accomplishments
7. Performance on the TPTP Problems
8. Bugs and Fixes
9. Otter-users Mailing List

## Related Pages

- Try Otter *right now* with **Son of BirdBrain**
- A sample Otter proof
- New Results obtained with Otter and related programs
- MACE, a program that searches for small models
- EQP, a prover for equational logic with associative unification
- Automated Reasoning at Argonne

## External Work

- Johan Belinfante's Set Theory Work with Otter
- Some other theorem provers
- Otter mode for Emacs (from Holger Schauer)
- GOAL, by Guoxiang Huang and Dale Myers
- A student project on Otter by Jackson Pauls

## Description

Our current automated deduction system Otter is designed to prove theorems stated in first-order logic with equality. Otter's inference rules are based on resolution and paramodulation, and it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing

100%

# Example: Robbins Algebras Are Boolean

- The Robbins problem---are all Robbins algebras Boolean?---has been solved: Every Robbins algebra is Boolean. This theorem was proved automatically by EQP, a theorem proving program developed at Argonne National Laboratory

# Example: Robbins Algebras Are Boolean

**Historical Background**

- In 1933, E. V. Huntington presented [1,2] the following basis for Boolean algebra:

- x + y = y + x. [commutativity]
- (x + y) + z = x + (y + z). [associativity]
- n(n(x) + y) + n(n(x) + n(y)) = x. [Huntington equation]

- Shortly thereafter, Herbert Robbins conjectured that the Huntington equation can be replaced with a simpler one [5]:
- n(n(x + y) + n(x + n(y))) = x. [Robbins equation]
- Robbins and Huntington could not find a proof, and the problem was later studied by Tarski and his students

# Example: Winker Conditions (1979)

- all x, n(n(x))=x
- $\exists$ 0 all x, x+0=x
- all x, x+x=x
- 1st: $\exists$ C $\exists$ D, C+D=C
- 2nd: $\exists$ C $\exists$ D, n(C+D)=n(C)

# Example: Otter: October 10, 1996

- n(n(n(y)+x)+n(x+y)) = x. [Robbins equation]
- n(x+y) != n(x). [denial of 2nd Winker condition]

The Robbins problem---are all Robbins algebras Boolean?---has been solved: Every Robbins algebra is Boolean. This theorem was proved automatically by EQP, a theorem proving program developed at Argonne National Laboratory.

## Historical Background

In 1933, E. V. Huntington presented [1,2] the following basis for Boolean algebra:

```
x + y = y + x.                          [commutativity]
 (x + y) + z = x + (y + z).             [associativity]
n(n(x) + y) + n(n(x) + n(y)) = x.       [Huntington equation]
```

Shortly thereafter, Herbert Robbins conjectured that the Huntington equation can be replaced with a simpler one [5]:

```
n(n(x + y) + n(x + n(y))) = x.          [Robbins equation]
```

Robbins and Huntington could not find a proof, and the problem was later studied by Tarski and his students [6].

```
Searching ...

Success, in 1.28 seconds!

---------------- PROOF ----------------

1          n(n(A)+B)+n(n(A)+n(B))!=A.                    []
2          x=x.                                          []
3          x+y=y+x.                                      []
5,4         (x+y)+z=x+ (y+z).                            []
6          n(n(x+y)+n(x+n(y)))=x.                        []
8          x+x=x.                                        []

10         n(n(A)+n(B))+n(n(A)+B)!=A.                    [para_from,3,1]
13         x+ (x+y)=x+y.                                 [para_into,4,8,flip.1]
15         x+ (y+z)=y+ (x+z).                            [para_into,4,3,demod,5]
23,22      x+ (y+x)=x+y.                                 [para_into,13,3]
26         n(n(x)+n(x+n(x)))=x.                          [para_into,6,8]
36         n(n(n(x)+x)+n(n(x)))=n(x).                    [para_into,6,8]
42         n(n(x+n(y))+n(x+y))=x.                        [para_into,6,3]
52         x+ (y+z)=x+ (z+y).                            [para_into,15,3,demod,5]
81,80      n(n(x+n(x))+n(x))=x.                          [para_into,26,3]
82         n(n(n(x)+x)+x)=n(x).                          [para_from,26,6,demod,23]
125        n(n(n(x+n(x))+ (n(x)+x))+x)=n(x+n(x))+n(x).   [para_into,80,80,demod,5,81]
139        n(n(n(x+n(x))+x)+x)=n(x+n(x)).                [para_from,80,6]
166,165    n(n(x+n(x))+x)=n(x).                          [para_into,82,3]
180,179    n(n(x)+x)=n(x+n(x)).                          [back_demod,139,demod,166]
195        n(n(x+n(x))+n(n(x)))=n(x).                    [back_demod,36,demod,180]
197        n(n(x+ (n(x)+n(x+n(x))))+ (n(x+n(x))+x))=n(x). [para_into,165,165,demod,5,180,5,166]
206,205    n(n(x+ (n(x)+n(x+n(x))))+n(x))=n(x+n(x))+x.   [para_from,165,80,demod,166,5,180,5]
223,222    n(n(x+y)+ (y+x))=n(x+ (y+n(x+y))).            [para_into,179,52,demod,5]
231,230    n(n(x+ (n(x)+n(x+n(x))))+x)=n(x+n(x))+n(x).   [back_demod,125,demod,223]
564,563    n(x+n(x))+x=x.                                [para_into,195,80,demod,5,223,81,206,81]
582,581    n(x+n(x))+n(x)=n(x).                          [back_demod,197,demod,564,231]
586,585    n(n(x))=x.                                    [back_demod,80,demod,582]
606,605    n(x+n(y))+n(x+y)=n(x).                        [para_into,585,42,flip.1]
621        A!=A.                                         [back_demod,10,demod,606,586]
622        $F.                                           [binary,621,2]

------------ end of proof --------------
```

**Given to the system**
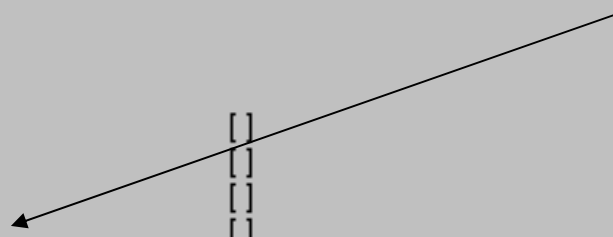
# Forward-chaining production systems

- Prolog & other programming languages: rely on backward-chaining

  (I.e., given a query, find substitutions that satisfy it)

- Forward-chaining systems: infer everything that can be inferred from KB each time new sentence is TELL'ed

- Appropriate for agent design: as new percepts come in, forward-chaining returns best action

# Implementation

- One possible approach: use a theorem prover, using resolution to forward-chain over KB

- More restricted systems can be more efficient.

- Typical components:
  - KB called "working memory" (positive literals, no variables)
  - rule memory (set of inference rules in form

    $$p1 \wedge p2 \wedge \ldots \Rightarrow act1 \wedge act2 \wedge \ldots$$

  - at each cycle: find rules whose premises satisfied

    by working memory (match phase)
  - decide which should be executed (conflict resolution phase)
  - execute actions of chosen rule (act phase)

# Match phase

- Unification can do it, but inefficient

- Rete algorithm (used in OPS-5 system): example

rule memory:

$A(x) \wedge B(x) \wedge C(y) \Rightarrow$ add $D(x)$

$A(x) \wedge B(y) \wedge D(x) \Rightarrow$ add $E(x)$
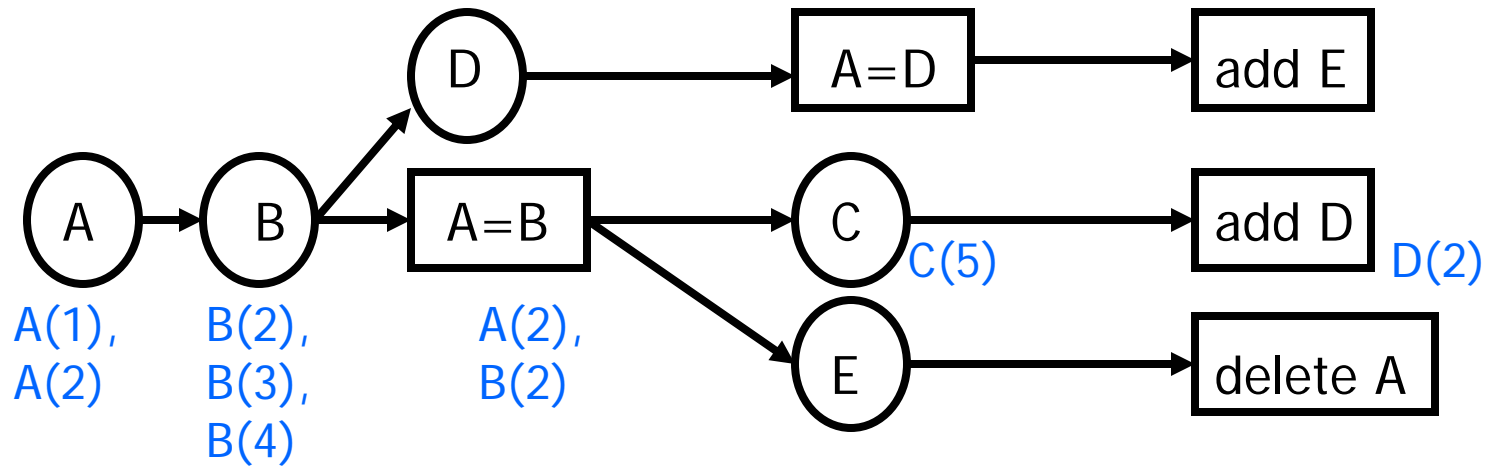
$A(x) \wedge B(x) \wedge E(x) \Rightarrow$ delete $A(x)$

working memory:

$\{A(1), A(2), B(2), B(3), B(4), C(5)\}$

- Build Rete network from rule memory, then pass working memory through it

# Rete network



Circular nodes: fetches to WM; rectangular nodes: unifications
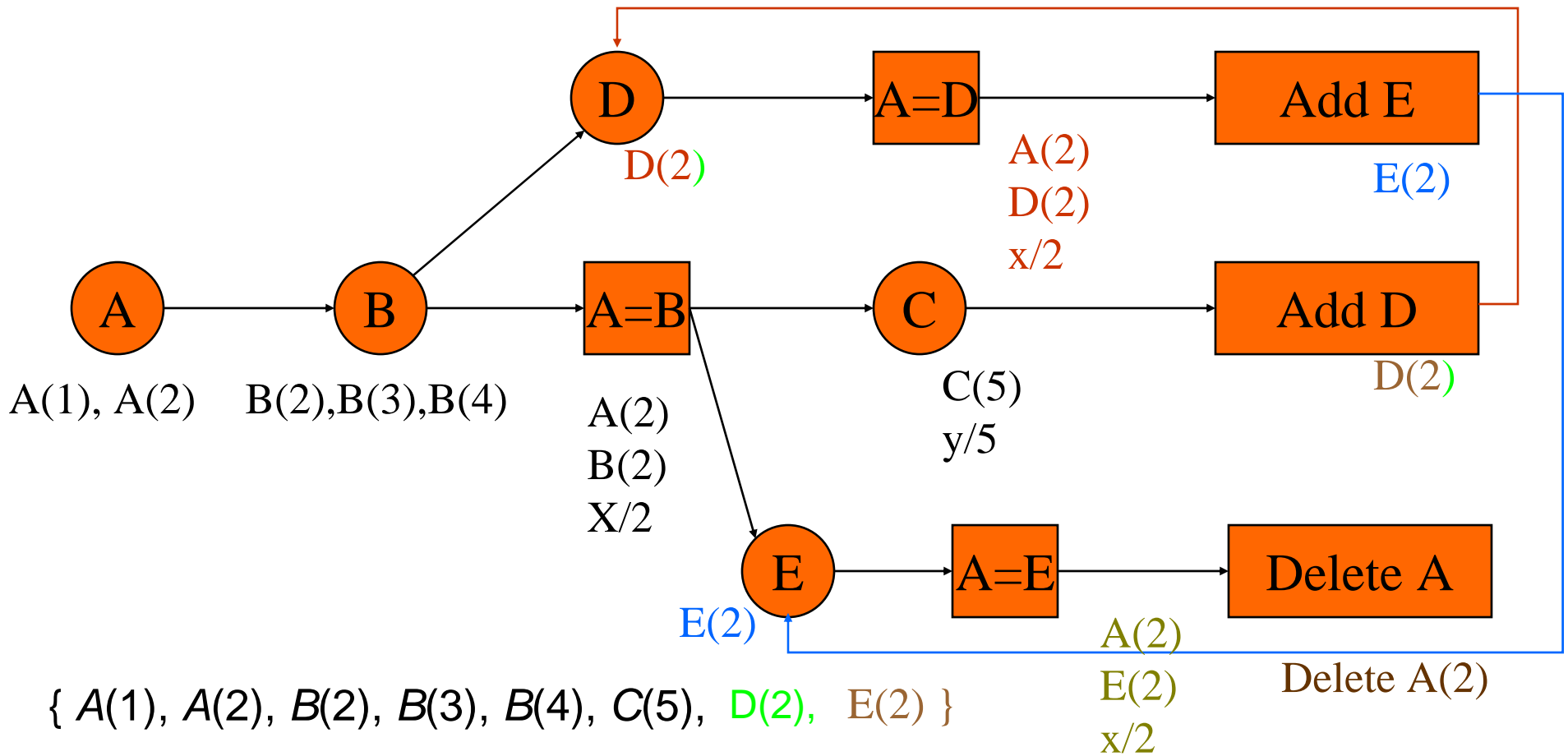
$A(x) \land B(x) \land C(y) \Rightarrow$ add D(x)

$A(x) \land B(y) \land D(x) \Rightarrow$ add E(x)

$A(x) \land B(x) \land E(x) \Rightarrow$ delete A(x)

{A(1), A(2), B(2), B(3), B(4), C(5)}

**Rete match**

$A(x) \wedge B(x) \wedge C(y) \Rightarrow$ add $D(x)$
$A(x) \wedge B(y) \wedge D(x) \Rightarrow$ add $E(x)$
$A(x) \wedge B(x) \wedge E(x) \Rightarrow$ delete $A(x)$



A(1), A(2)    B(2),B(3),B(4)

D(2)

A=D
A(2)
D(2)
x/2

Add E
E(2)

A=B
A(2)
B(2)
X/2

C
C(5)
y/5

Add D
D(2)

E(2)

E
A=E
A(2)
E(2)
x/2

Delete A
Delete A(2)

{ $A(1)$, $A(2)$, $B(2)$, $B(3)$, $B(4)$, $C(5)$, $D(2)$, $E(2)$ }

## Advantages of Rete networks

- Share common parts of rules

- Eliminate duplication over time (since for most production systems only a few rules change at each time step)

# Conflict resolution phase

- one strategy: execute all actions for all satisfied rules

- or, treat them as suggestions and use conflict resolution to pick one action.

- Strategies:
    - no duplication (do not execute twice same rule on same args)
    - regency (prefer rules involving recently created WM elements)
    - specificity (prefer more specific rules)
    - operation priority (rank actions by priority and pick highest)

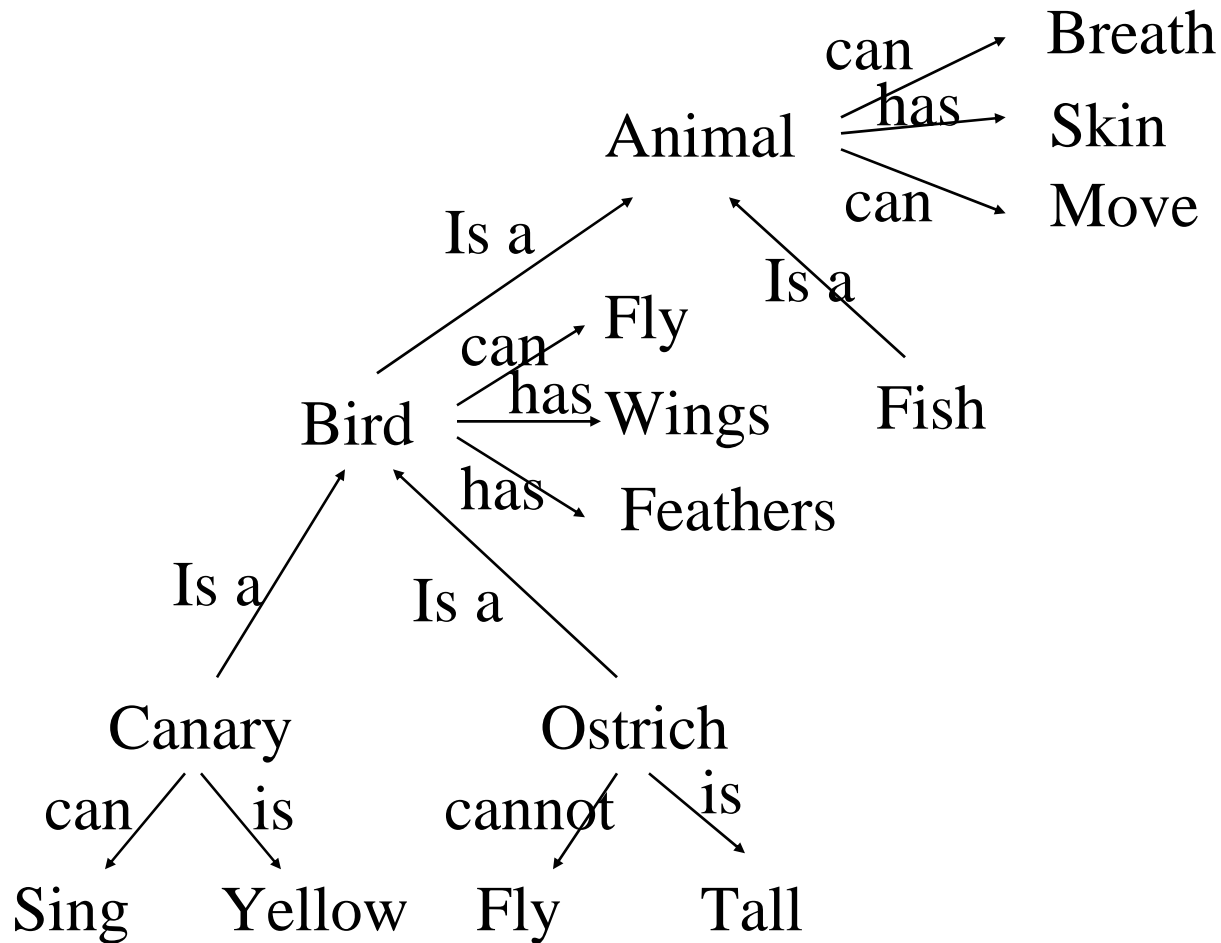# Frame systems & semantic networks

- Other notation for logic; equivalent to sentence notation

- Focus on categories and relations between them (remember ontologies)

- e.g., Cats $\xrightarrow{\textit{Subset}}$ Mammals

# Syntax and Semantics

| Link Type | Semantics |
|-----------|-----------|
| $A \overset{\text{Subset}}{\rightarrow} B$ | $A \subset B$ |
| $A \overset{\text{Member}}{\rightarrow} B$ | $A \in B$ |
| $A \overset{R}{\rightarrow} B$ | $R(A,B)$ |
| $A \overset{\boxed{R}}{\Rightarrow} B$ | $\forall x \; x \in A \Rightarrow R(x,y)$ |
| $A \overset{\boxed{\boxed{R}}}{\rightarrow} B$ | $\forall x \; \exists y \; x \in A \Rightarrow y \in B \wedge R(x,y)$ |

# Semantic Network Representation



Animal can → Breath
Animal has → Skin
Animal can → Move

Bird Is a Animal
Fish Is a Animal

Bird can → Fly
Bird has → Wings
Bird has → Feathers

Canary Is a Bird
Ostrich Is a Bird

Canary can → Sing
Canary is → Yellow

Ostrich cannot → Fly
Ostrich is → Tall

# Semantic network link types

| Link type | Semantics | Example |
|---|---|---|
| A $\xrightarrow{Subset}$ B | A $\subset$ B | Cats $\xrightarrow{Subset}$ Mammals |
| A $\xrightarrow{Member}$ B | A $\in$ B | Bill $\xrightarrow{Member}$ Cats |
| A $\xrightarrow{R}$ B | R(A, B) | Bill $\xrightarrow{Age}$ 12 |
| A $\xrightarrow{\boxed{R}}$ B | $\forall x \quad x \in A \Rightarrow R(x, B)$ | Birds $\xrightarrow{\boxed{Legs}}$ 2 |
| A $\xrightarrow{\boxed{\boxed{R}}}$ B | $\forall x \exists y \; x \in A \Rightarrow y \in B \wedge R(x, y)$ | Birds $\xrightarrow{\boxed{\boxed{Parent}}}$ Birds |

# Description logics

- FOL: focus on objects

- Description logics: focus on categories and their definitions

- Principal inference tasks:
  - subsumption: is one category subset of another?
  - classification: object belings to category?

# CLASSIC

- And(concept, ...)
- All(RoleName, Concept)
- AtLeast(Integer, RoleName)
- AtMost(Integer, RolaName)
- Fills(RoleName, IndividualName, ...)
- SameAs(Path, Path)
- OneOf(IndividualName, ...)


e.g., Bachelor = And(Unmarried, Adult, Male)