1988

Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

```
delim %%

predicate expression



predicate               arguments
(alias predicate
name)



ghwizhge(ggirjajdj,365,ajgc52)



first                   second                  third
argument                argument                        argument
```

Figure 2-1: terminology about predicate expressions

|type|property|relationship|database|function|probability
|predicate|predicate|predicate|predicate|predicate|predicate

```
Number|1|2|2|1 or more|2 or more|1 or more
of
arguments
Nature|a thing|a thing and|two things|a thing|last is|last is
of||a property||and|result of|probability
arguments||||properties|operation|of the
|||||on others|fact truth
Description|gives a class|gives|describes|like a|describes|variant of
|the thing|property of|relationship|data record|a function|preceding
|belongs to|a thing|of two||mapping|for partly
|||things|||certain facts
Examples|ship|color|part_of|ship|sum|color
|(kennedy)|(kennedy,|(kennedy,|(kennedy,|(3,|(kennedy,
||gray)|u_s_navy)|gray,|4,|gray,
|vehicle|||14n35e,|7)|0.8)
|(ship)|location|a_kind_of|021685)
||(kennedy,|(kennedy,
||14n35e)|ship)|
Meaning|"The Kennedy|"The color of|"The Kennedy|"There is|"The sum|"We believe
of the|is a ship"|the Kennedy|is part of the|a ship|of 3 and|with certainty
examples||is gray"|U.S. Navy"|record with|4 is 7"|0.8 that the
|"A ship is|||entries||Kennedy
|a vehicle"|"The location|"The Kennedy|Kennedy,||is gray"
||of the Kennedy|is a kind|gray, and
||is 14n35e"|of ship"|14n35e"
```

Figure 2-2: summary of the kinds of predicates in Chapter 2

```
gray                                    government

        color                                   a_kind_of

ship                                    u_s_government

                                                        has
        a_kind_of                               part_of
                a_kind_of

enterprise              kennedy                 u_s_navy

                                part_of                 civil_service_system
```

```
                part_of

        location

15n35e
```

Figure 2-3: an example semantic network

```
Symbol|Description|Meaning
?-|question mark,|query prompt
|minus sign|(typed by Prolog)
.|period|(1) query terminator (typed by user)
||(2) database fact and rule terminator
, |comma|(1) "and" of predicate expressions in a query
||(2) separator of arguments
||(3) separator of list items (see chapter 5)
;|semicolon|(1) "or" of predicate expressions in a query
||(2) forces new alternative after a query answer
()|parentheses|(1) for grouping arguments together
||(2) for grouping query expressions
_ |underscore|no special meaning, but we often use
||as a character to make long names legible
:- |colon,|rule definition symbol (see
|minus sign|chapter 4); read as "if"
[] |square brackets|a list (see chapter 5)
<> |angular brackets|not a Prolog symbol, but
||we use it to enclose descriptions of
||things (like Backus-Naur Form)
```

Figure 3-2: special symbols used with Prolog in this book

Query:

```
?-boss(X,Y), boss(Y,Z).
```

Database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Processing:

```
Step number|First boss predicate expression:|Second boss predicate expression:
in text|Bindings made|Binding of Z
|of X and Y

1|X=dick,Y=harry|
2||[no Z possible]
3|X=tom,Y=dick|
4||Z=harry
5||
```

Figure 3-3: processing state at generation of first answer to superboss query: X=tom,Y=dick,Z=harry

---

Query:

```
?-boss(X,Y), boss(Y,Z).
```

Database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Processing:

```
Step number|First boss predicate expression:|Second boss predicate expression:
in text|Bindings made|Binding of Z
|of X and Y
1|X=dick,Y=harry|
2||[no Z possible]
3|X=tom,Y=dick|
4||Z=harry
5||
6||
7||[no further Z possible]
8|X=ann,Y=mary|
9||Z=harry|
10||
```

Figure 3-4: processing state at generation of second answer to superboss query: X=ann,Y=mary,Z=harry

---

Query:

```
?-boss(X,Y), not(boss(X,dick)), boss(Y,Z).
```

Database:

```
boss(dick,harry).
boss(tom,dick).
boss(ann,mary).
boss(mary,harry).
```

Processing:

```
Step number|First predicate|Second predicate|Third predicate
in text|expression:|expression:|expression:
|Bindings made|What happens?|Binding of Z
|of X and Y
1|X=dick,Y=harry
2||succeeds
3|||[no possible Z]
4||fails
4|X=tom,Y=dick
5||fails|
6|X=ann,Y=mary
7||succeeds
8|||Z=harry
```

Figure 3-5: processing order for the extended superboss example: answer X=ann,Y=mary,Z=harry

---

Database:

```
Memory location|Contents
10000|a_kind_of(enterprise,ship).
10026|a_kind_of(kennedy,ship).
10052|part_of(enterprise,u_s_navy).
10070|part_of(kennedy,u_s_navy).
10088|part_of(u_s_navy,u_s_government).
```

Index:

```
Predicate name|Locations of facts using it

a_kind_of|10000,10026
part_of|10052,10070,10088
```

Figure 3-6: an example of predicate indexing

```
left side                    special symbol          right side
(a predicate            meaning "if"            (a query)
expression)
```

```
a(X,middle,Y) :- b(X), c(X,Y,extra), d(X,Z), e(Z,Y), f(Z).


parameter variables             local variable          rule terminator
```

Read a rule as meaning: the left side is true if querying the right side succeeds. Figure 4-1: terminology about rules

---

Query:

```
?-boss(X,Y), not(boss(X,tom)).
```

Database:

```
department(tom,sales).
department(harry,production).
manager(dick,sales).
manager(mary,production).
boss(B,E) :- department(E,D), manager(B,D).
```

Processing:

```
Step|In rule|In rule|In query:|In rule|In rule|In query:
in text|call 1:|call 1:|Bindings|call 2:|call 2:|argument
|Bindings|Binding|to X and Y|Bindings|Binding|to not
|to E and D|to B||to B and E|to D|

1|E=tom,
|D=sales
2||B=dick|
2|||X=dick,
|||Y=tom
3||||
4||||B=dick,
||||E=tom
4|||||D=sales
4||||||succeeds
5||[no more
||choices]
6|E=harry,
|D=production
7||B=mary
7|||X=mary,
|||Y=harry
8||||B=mary,
||||E=tom
9|||||D=sales
10||||||fails
```

Figure 4-2: processing order of the other-than-tom's-boss query: answer X=mary,Y=harry

---

```
X       r       Z       r       Y

                r
```

Figure 4-3: the general form of transitivity

```
                d

                c

                b

                a


on(b,a).
on(c,b).
on(d,c).
```

Figure 4-4: a transitivity example

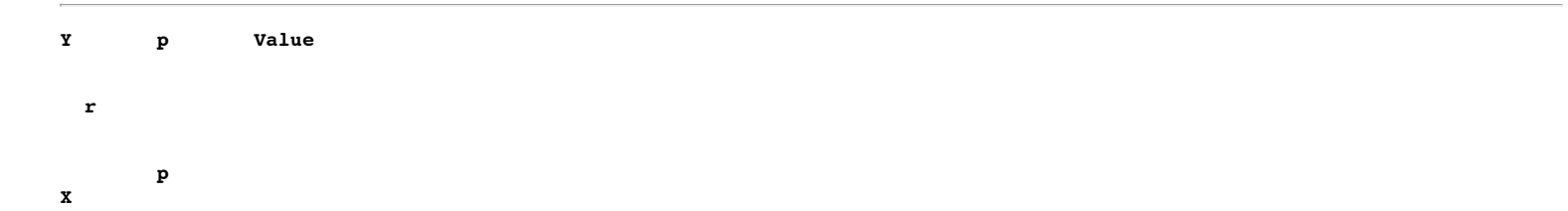


Figure 4-5: the general form of inheritance

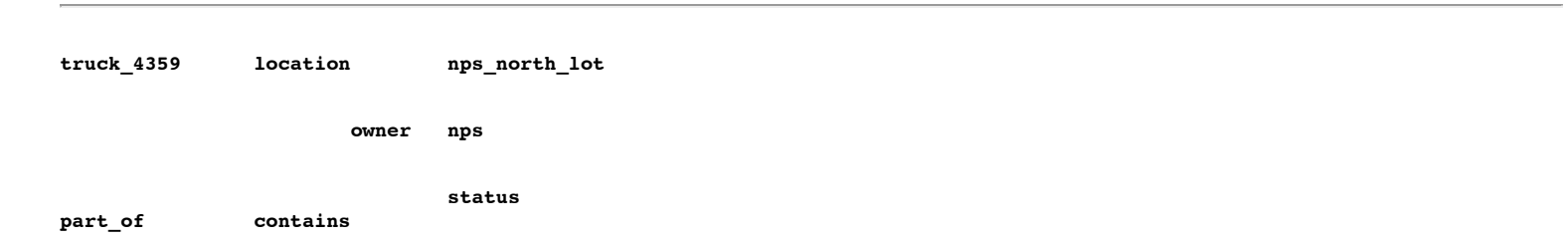


Figure 4-6: an inheritance example

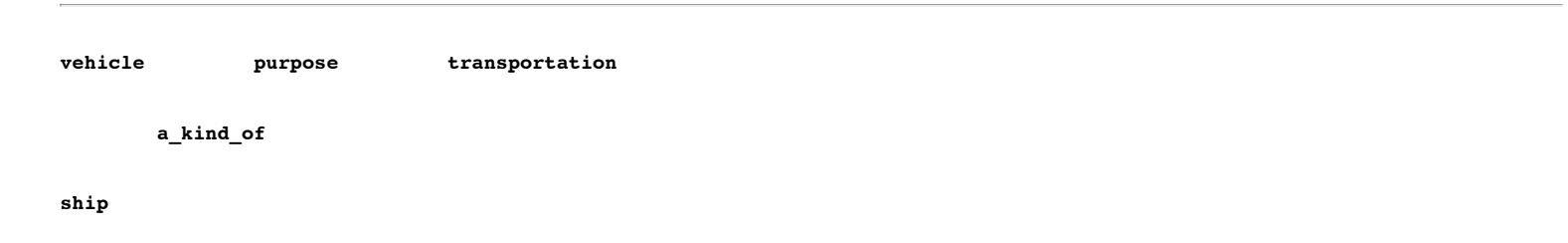
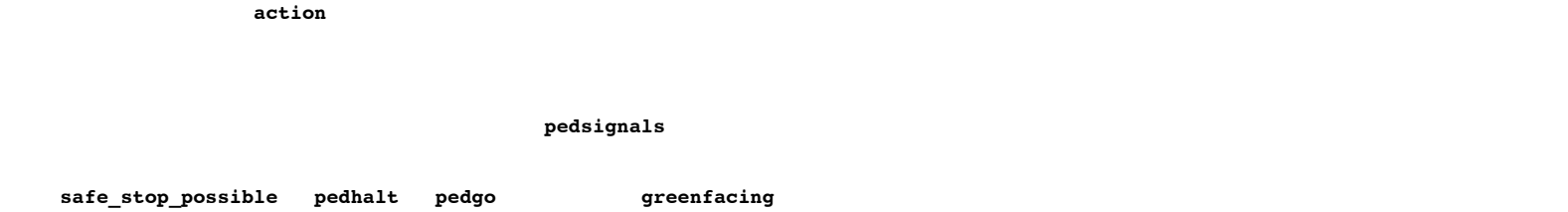


Figure 4-7: an example of both transitivity and inheritance



```
                light                          clockwise_cross
```

Figure 4-8: the predicate hierarchy for the traffic lights program

---

```
 D
 C      G
 B      F   I
 A      E   H
---------------------------
```

Figure 4-9: some blocks on a table

```
Predicate|Arguments|What it means

first(L,I)|a list,|I is the first
|an item|item of the list L
last(L,I)|a list,|I is the last
|an item|item of the list L
member(I,L)|an item,|I is an item somewhere
|a list|inside the list L
length(L,N)|a list,|N is the length
|a number|of the list L
max(L,N)|a list,|N is the largest number
|a number|in the list of numbers L
delete(I,L1,L2)|an item,|L2 is L1 with all
|a list,|occurrences of the I
|a list|removed from it
append(L1,L2,L3)|a list,|L3 is the glueing
|a list,|together of the
|a list|lists L1 and L2
subset(L1,L2)|a list,|all items in L1
|a list,|are in L2
sort(L1,L2)|a list,|L2 is the sorting
|a list|of list of numbers L1
```

Figure 5-1: the classic list-processing predicates defined in this chapter (to be used in the rest of this book)

Given the definition of append:

```
append([],L,L).
append([X|L],L2,[X|L3]) :- append(L,L2,L3).
```

Given the query:

```
?- append([gas,oil],[tires,battery,radiator],Things_to_check_before_trip).
```

Nested environments created:

```
Query: ?- append([gas,oil],[tires,battery,radiator],Things_to_check_before_trip).
In definition line 2: [X|L]=[gas,oil], L2=[tires,battery,radiator], [X|L3]=???
So X=gas, L=[oil], [X|L3]=[gas|???]


        Query: ?- append([oil],[tires,battery,radiator],L3).
        In definition line 2: [X|L]=[oil], L2=[tires,battery,radiator], [X|L3]=???
        So X=oil, L=[], [X|L3]=[oil|L3]


                Query: ?- append([],[tires,battery,radiator],L3).
                In definition line 1: L=[tires,battery,radiator]
                Query succeeds
```

Note: "???" means unknown Figure 5-2: an example of using append with first two arguments bound (inputs) and last argument unbound (an output); the processing state is just after the innermost recursion succeeds

---

Given the definition of append:

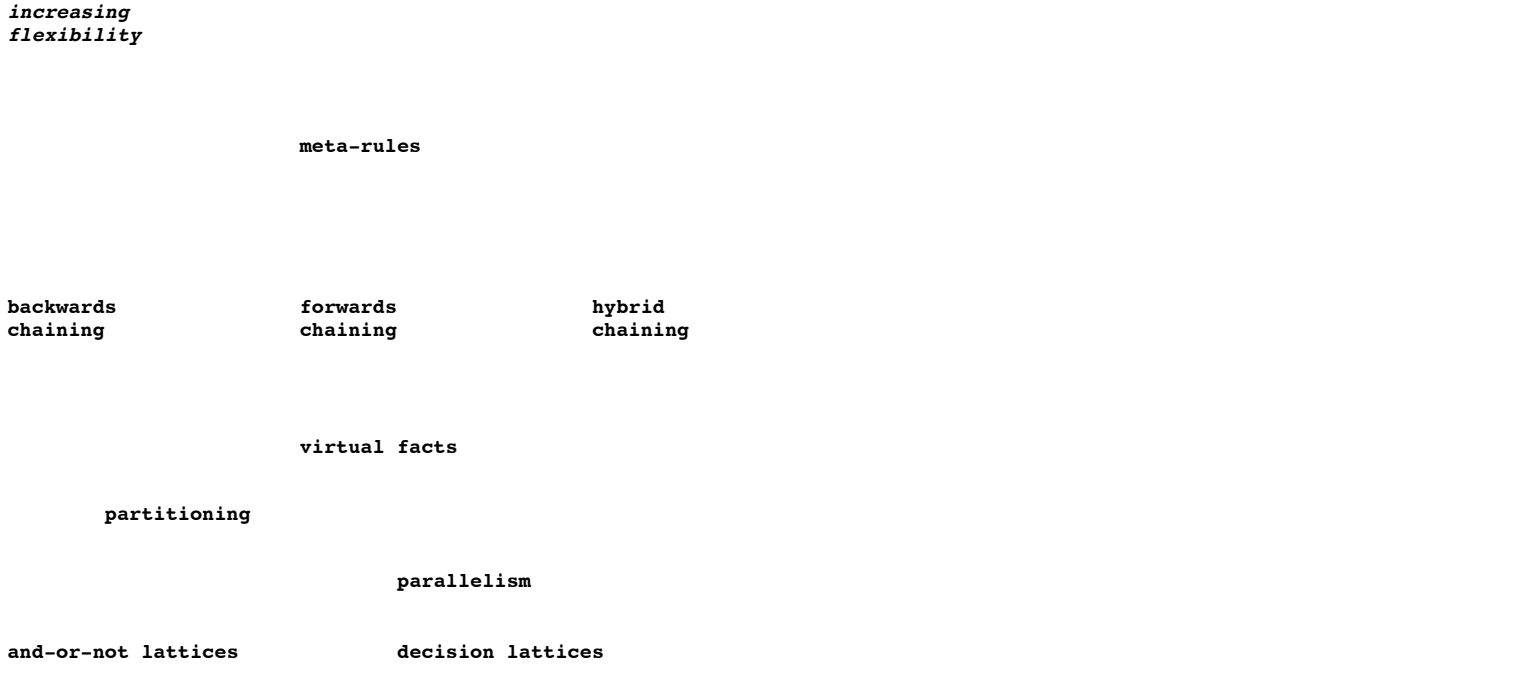
```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

first arg.	*second arg.*	*third arg.*	*What happens?*	*Example*
bound	bound	bound	Answers yes if third argument is the first two arguments glued together (in order), else no	?- append([a,b],[c],[a,b,c]).
bound	bound	unbound	Glues the second argument on the end of the first argument, binds result to the third argument	?- append([a,b],[c,d],L). L = [a,b,c,d]
bound	unbound	bound	Checks if the first argument is on the, front of the third, binding the second to the rest of the third	?- append([a,b],L,[a,b,c]). L = [c]
unbound	bound	bound	Checks if the second argument is on the back of the third, binding the first to the rest of the third	?- append(L,[c,d],[a,b,c,d]). L = [a,b]
unbound	unbound	bound	Generates all divisions of the third argument into two pieces (preserving term order)	?- append(L1,L2,[a,b,c]). L1 = [], L2 = [a,b,c]; L1 = [a], L2 = [b,c]; L1 = [a,b], L2 = [c]; L1 = [a,b,c], L2 = []
unbound	bound	unbound	Generates in abstract form all results of glueing something on the front of the second argument	?- append(L1,[a,b],L2). L1 = [], L2 = [a,b]; L1 = [_1], L2 = [_1,a,b]; L1 = [_1,_2], L2 = [_1,_2,a,b] (The _1 and _2 are Prolog-invented variable names.)
bound	unbound	unbound	Generates in abstract form all results of glueing something on the back of the first argument	?- append([a,b],L1,L2). L1 = _1, L2 = [a,b|_1] (The _1 is a Prolog-invented variable name.)

Note: any bound arguments to append must be lists for the above program to work.

Figure 5-3: the different possible uses of the append predicate

---



```
                    caching




increasing
efficiency
```

Figure 6-1: the spectrum of control-structure concepts for rule-based systems

---

```
Step number|Which fact|Which rule|What happens?
|being pursued?|matched?

1|fact2|R8|Define R10: d :- fact3.
|fact2|R9|Define R11: e :- fact4.
2|fact3|R10|New fact found: d.
3|d|R4|nothing
|d|R5|New fact found: b.
4|b|R2|Define R12: goal1 :- a.
5|--|--
6|not(d)|R4|nothing, since d a fact
|not(e)|R7|New fact found: c.
7|c(2)|R3|New fact found: goal2(2).
```

Figure 6-2: summary of the example of pure forwards chaining

```
Pure backwards chaining:|Pure forwards chaining:|Rule-cycle hybrid:



query order|fact order|rule order

has priority over|has priority over|has priority over

rule order|rule order|rule-right-side order

which has priority over|which has priority over|which has priority over

rule-right-side order|rule-right-side order|fact order

which has priority over

fact order
```

Figure 6-3: ordering priorities for three kinds of chaining

```
Cycle no.|R1|R2|R3|R4|R5|R6|R7|R8|R9

1|fails|fails|fails|skipped|fails|fails|skipped|succeeds|fails
2|fails|fails|fails|skipped|succeeds|--|skipped|--|fails
3|fails|fails|fails|skipped|--|--|skipped|--|fails
4|fails|fails|fails|fails|--|--|succeeds|--|fails
5|fails|fails|succeeds|--|--|--|--|--|--
```

Figure 6-4: summary of the example of rule-cycle hybrid chaining (the "--" means not necessary to test)

```
                    appliance works?

              no                    yes

       plugged in?                  what kind of appliance?

       no      yes                         heating         mechanical      other
```

```
plug it in      are nearby electrical          [further       [further       [further
and stop        appliances working?            questions]     questions]     questions]

                no              unclear        yes

                        plug into different
                        outlet; work now?

                no                     yes

blown fuse or                          stop
circuit breaker?

yes             no
                                               [must be internal problem,
                                               so ask more about this]
fix fuse or
circuit breaker
and stop
```

Figure 6-5: a decision lattice

```
        and parallelism
                                    partition parallelism
a(X) :- b, c(X), d(X,Y).                    r(X) :- s(X), t(X,Y).
                                                r(X) :- u(X).


        or parallelism



a(X,Y) :- e(X,Y), f(X,Y), not(c(X)).



        variable-matching parallelism
```

Figure 6-6: types of parallelism

```
question

noun_phrase linker modifier ?

modifier noun prepositional_phrase linker modifier ?

which noun prepositional_phrase linker modifier ?

which ships prepositional_phrase linker modifier ?

which ships preposition noun_phrase linker modifier ?

which ships in noun_phrase linker modifier ?

which ships in the noun linker modifier ?
```

```
which ships in the Mediterranean linker modifier ?

which ships in the Mediterranean are modifier ?

which ships in the Mediterranean are American ?
```

Figure 6-8: a parsing example (downwards arrows represent top-down parsing, upwards arrows represent bottom-up)

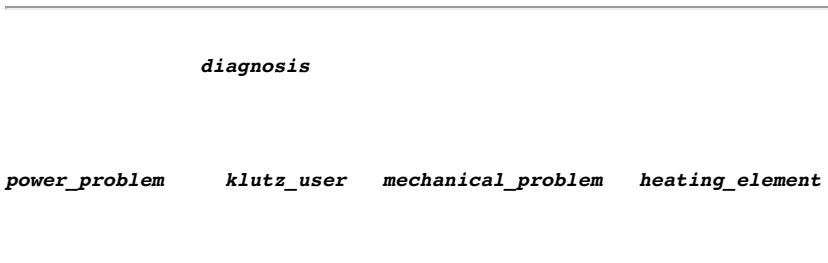


Figure 7-2: the predicate hierarchy for the appliance diagnosis expert system, including the problem-independent predicates

---

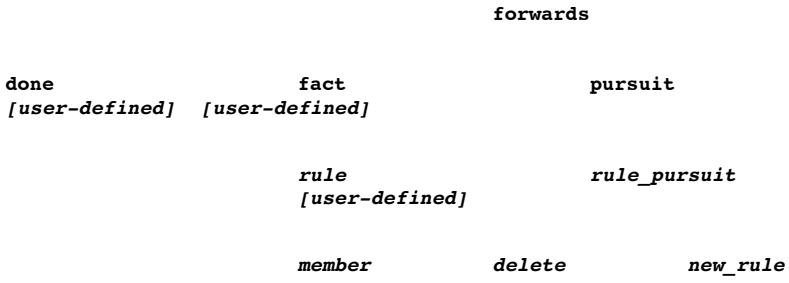
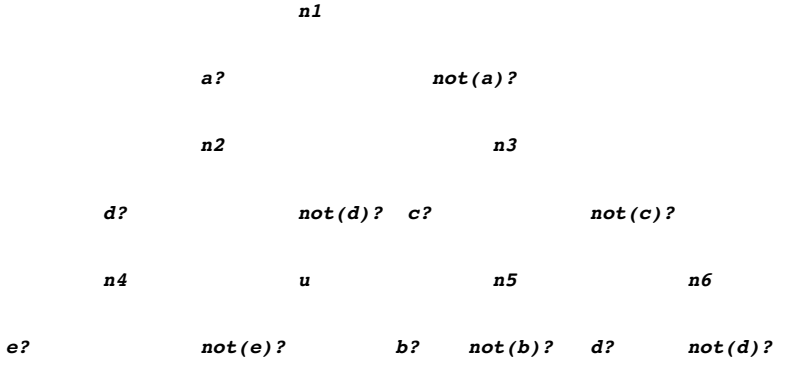


Figure 7-3: the predicate hierarchy for the forwards-chaining program

---



u *r* *t* *v* *t* *s*

Figure 7-4: a derived decision tree

```
Independence-assumption "or":
        p(A or B) = p(A) + p(B) - p(A)p(B)
Independence-assumption "and":
        p(A and B) = p(A)p(B)



Conservative "or":
        p(A or B) = p(A) if p(A)>p(B), else p(B)
Liberal "and":
        p(A and B) = p(B) if p(A)>p(B), else p(A)



Liberal "or":
        p(A or B) = p(A)+p(B) if this < 1, else 1
Conservative "and":
        p(A and B) = p(A)+p(B)-1 if this > 0, else 0
```

```
                                                                p(A)            p(B)
```

Figure 8-2: Venn diagrams illustrating the three standard probability-combination methods, applied to two probabilities

---

|and-combination|or-combination

```
Independence assumption|%p sub 1 p sub 2 ... p sub n%|%1 - [ ( 1 - p sub 1 ) ( 1 - p sub 2 ) ... ( 1 - p sub n ) ]%
(reasonable-guess)
Conservative assumption|%max ( 0 , ( p sub 1 + p sub 2 + ... + p sub n ) - n + 1 )%|%max ( p sub 1 , p sub 2 , ... p sub n )%
(lower bound)
Liberal assumption|%min ( p sub 1 , p sub 2 , ... p sub n )%|%min ( 1 , p sub 1 + p sub 2 + ... + p sub n ) %
(upper bound)
```

Note:

> **conservative "and" % <= % independence-assumption "and" % <= % liberal "and"**
> **% <= % conservative "or" % <= % independence-assumption "or" % <= % liberal "or"**

Figure 8-3: formulas for combination of probabilities %p sub i%, i=1 to n

---

```
|definitely|probably|probably not|definitely not|
definitely|definitely|definitely|definitely|definitely
probably|definitely|probably|probably|probably
probably not|definitely|probably|probably not|probably not|
definitely not|definitely|probably|probably not|definitely not|
```

Figure 8-5: one evidence combination method

```
|definitely|probably|probably not|definitely not|
definitely|definitely|definitely|definitely|definitely
probably|definitely|definitely|definitely|probably
probably not|definitely|definitely|probably|probably not|
definitely not|definitely|probably|probably not|definitely not|
```

Figure 8-6: another evidence combination method

```
        b       c
a                       e       f
```

Figure 9-2: the equivalent search graph for Figure 9-1

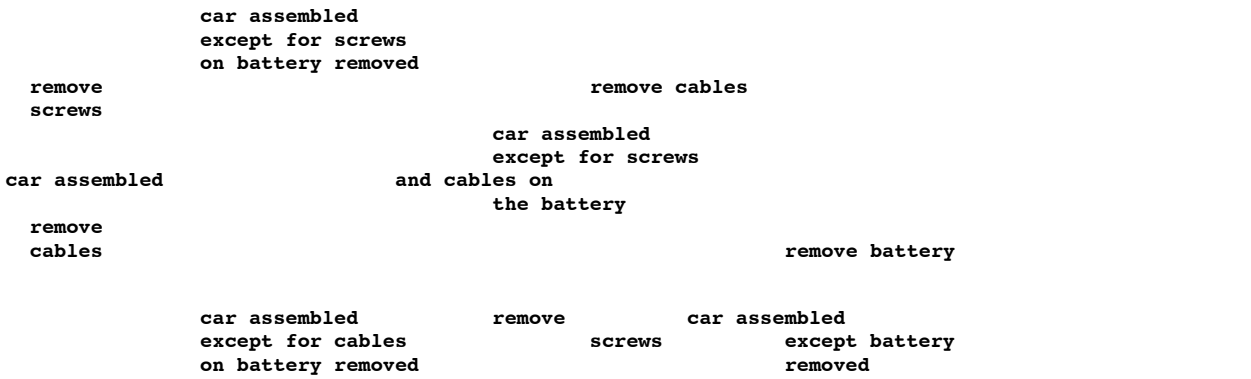


Figure 9-3: part of a search graph for auto repair

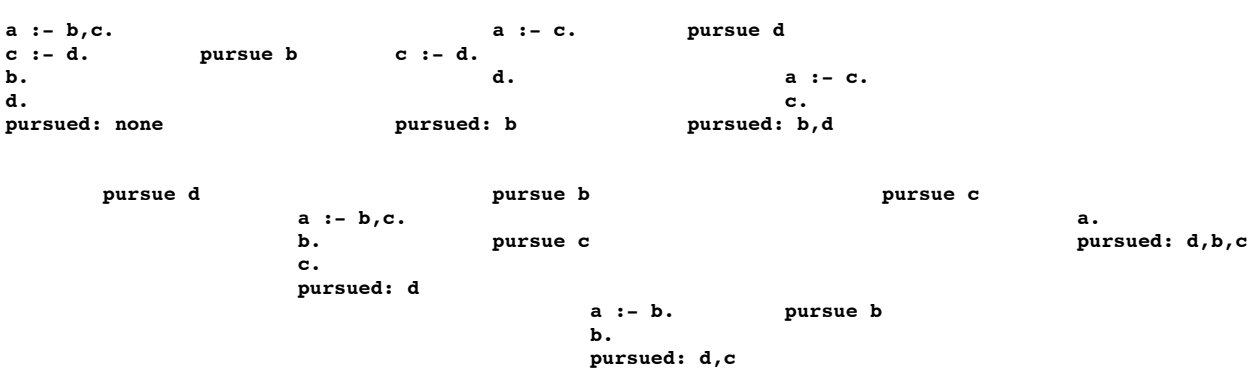


Figure 9-4: a complete search graph for generalized (any-fact-order) forwards chaining from a particular set of facts and rules (rules no longer useful are deleted from states as search proceeds)

Name of search strategy	*Uses agenda?*	*Uses evaluation function?*	*Uses cost function?*	*Next state whose successors are found*
Depth-first	no	no	no	A successor of the last state, else the predecessor
Breadth-first	yes	no	no	The state on the agenda the longest
Hill-climbing (optimization)	no	yes	no	The lowest-evaluation successor of the last state
Best-first search	yes	yes	no	The state on the agenda of lowest evaluation value
Branch-and-bound	yes	no	yes	The state on the agenda of lowest total cost
A* search	yes	yes	yes	The state on the agenda of lowest sum of evaluation value and total cost

Figure 9-5: the classic search strategies (heuristics may be used with any of these)

Given the rule set:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

Then this is a decision lattice implementing it:

```
                        a?


        yes                     no


        b?                      c?


yes             no      yes             no


conclude        c?      conclude                b?
t                       t


        yes     no                      yes             no


conclude        d?                      e?              fail
t
        yes             no              yes     no


        conclude        fail            conclude        fail
        u                               v
```

Figure 9-10: the decision lattice for backwards chaining on an example rule set

---

Assume the rules:

```
t :- a, b.
t :- c.
u :- not(a), c.
u :- a, d.
v :- b, e.
```

And assume the only possible facts are a, b, c, d, and e, provided for a database in order of priority a, c, b, d, and e. Assume each has an independent probability of occurrence P. Situations to consider:

```
a?|c?|b?|d?|e?|probability|number of|conclusion
||||||matches|reached

false|true|-|-|-|%P ( 1 - P )%|1|t
true|true|-|-|-|%P sup 2%|3|t
true|false|true|-|-|%P sup 2 ( 1 - P )%|3|t
true|false|false|true|-|%P sup 2 ( 1 - P ) sup 2%|3|u
true|false|false|false|true|%P sup 2 ( 1 - P ) sup 3%|4|-
true|false|false|false|false|%P sup ( 1 - P ) sup 4%|3|-
false|false|true|true|true|%P sup 3 ( 1 - P ) sup 2%|4|v
false|false|true|true|false|%P sup 2 ( 1 - P ) sup 3%|4|-
false|false|true|false|true|%P sup 2 ( 1 - P ) sup 3%|3|v
false|false|true|false|false|%P ( 1 - P ) sup 4%|3|-
false|false|false|true|true|%P sup 2 ( 1 - P ) sup 3 %|3|-
false|false|false|true|false|%P ( 1 - P ) sup 4%|2|-
false|false|false|false|true|%P ( 1 - P ) sup 4%|2|
false|false|false|false|false|%( 1 - P ) sup 5%|1|-
```

Figure 9-11: analysis of cost of forwards chaining on the rule set

---

Given this array of edgeness measures for a picture (array g(i,j)):

```
0|1|0|8|2
1|2|7|3|1
3|1|5|2|0
2|2|3|3|1
```

```
2|5|2|1|3
1|7|1|2|1
3|6|2|2|1
```

Inference: there's a continuous edge running north-northeast to south-southwest, though it's hard to see towards the center of the picture. That's the interpretation (array e(i,j)):

```
false|false|false|true|false
false|false|true|false|false
false|false|true|false|false
false|false|true|false|false
false|true|false|false|false
false|true|false|false|false
false|true|false|false|false
```

Figure 9-13: example arrays for finding visual edges

```
                b               c


a


                d               e
```

Figure 10-3: a simpler route-planning problem

```
Step number|depthsearch2|depthsearch2|depthsearch2|depthsearch2
in text|first call|second call|third call|fourth call

1|called with
|state a
2|first rule
|fails
3|second|called with
|rule tried|state b
4||first rule|called with
||fails,|state c
||second tried
5|||both rules
|||fail on c
6||backtrack to
||not, then to
||successor;
||choose d
7|||called with
|||state d;
|||first rule
|||fails; 2nd
|||rule picks
|||b, which
|||fails not
8|||choose e|called with
||||state e
9||||first rule
||||succeeds
||||with path
||||list [e,d,b,a]
|||succeeds
||succeeds
|succeeds
```

Figure 10-4: summary of the depth-first search example

```
                breadthsearch


find_successors          cleandatabase           measurework
```

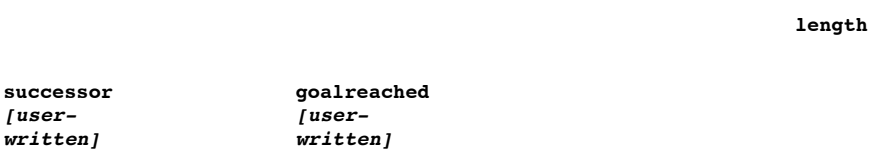


Figure 10-5: the rule-predicate hierarchy for the breadth-first search (breadthsearch) **program**

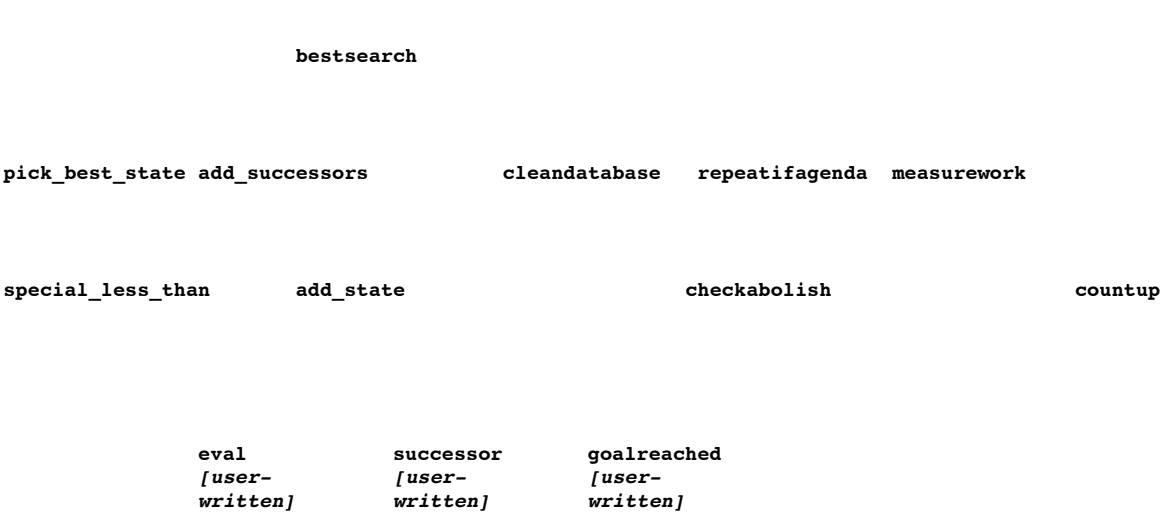


Figure 10-6: the rule-predicate hierarchy for the best-first search (bestsearch) **program**

Figure 10-7: the rule-predicate hierarchy for the A* search (astarsearch) program

---

Desired|Should|Should|Should|Should|Should
situation|you|you|you|you|you
|visit|visit|visit|visit|visit
|cafeteria?|vending|your|secretary?|colleague?
||machine?|office?

```
You want|yes|yes|no|no|no
to be not|
hungry,
and you're
hungry now
You want|no|no|yes|yes|no
to have
change,
and you
don't now
You want|no|no|yes|yes|yes
to know|
where
something
is
```

Figure 11-1: tabular representation of the operator recommendations (difference table) for the office-worker problem

---

```
Desired|Should|Should|Should|Should|Should
situation|you|you|you|you|you
|visit|visit|visit|visit|visit
|vending|cafeteria?|colleague?|secretary?|your|
|machine?||||office?

You want|yes|yes|no|no|no
to be not|
hungry,
and you're
hungry now
You want|no|no|no|yes|yes
to have
change,
and you
don't now
You want|no|no|yes|yes|yes
to know||||(but try|(but try
where||||third)|second)
something
is
```

Figure 11-2: a different tabular representation of the operator recommendations (difference table) for the office-worker problem

---

```
Desired|Should|Should|Should|Should|Should|Should|Should|Should
situation|you|you|you|you|you|you|you|you
|replace|replace|disassemble|disassemble|assemble|assemble|turn over|smash
|batteries|light|case|top|case|top|case|case
|?|?|?|?|?|?|?|?

You want|yes|no|no|no|no|no|no|no
batteries
OK, when
they aren't
You want|no|yes|no|no|no|no|no|no
light OK,
when it
isn't
You want|no|no|yes|no|no|no|no|yes
case open,
when it
isn't
You want|no|no|no|yes|no|no|no|yes
top open,
when it
isn't
You want|no|no|no|no|yes|no|no|no
case closed,
when it
isn't
You want|no|no|no|no|no|yes|no|no
top closed,
when it
isn't
You want|no|no|no|no|no|no|yes|yes
batteries
```

```
outside,
when they
aren't
```

Figure 11-4: tabular representation of the operator recommendations (difference table) for the flashlight problem

---

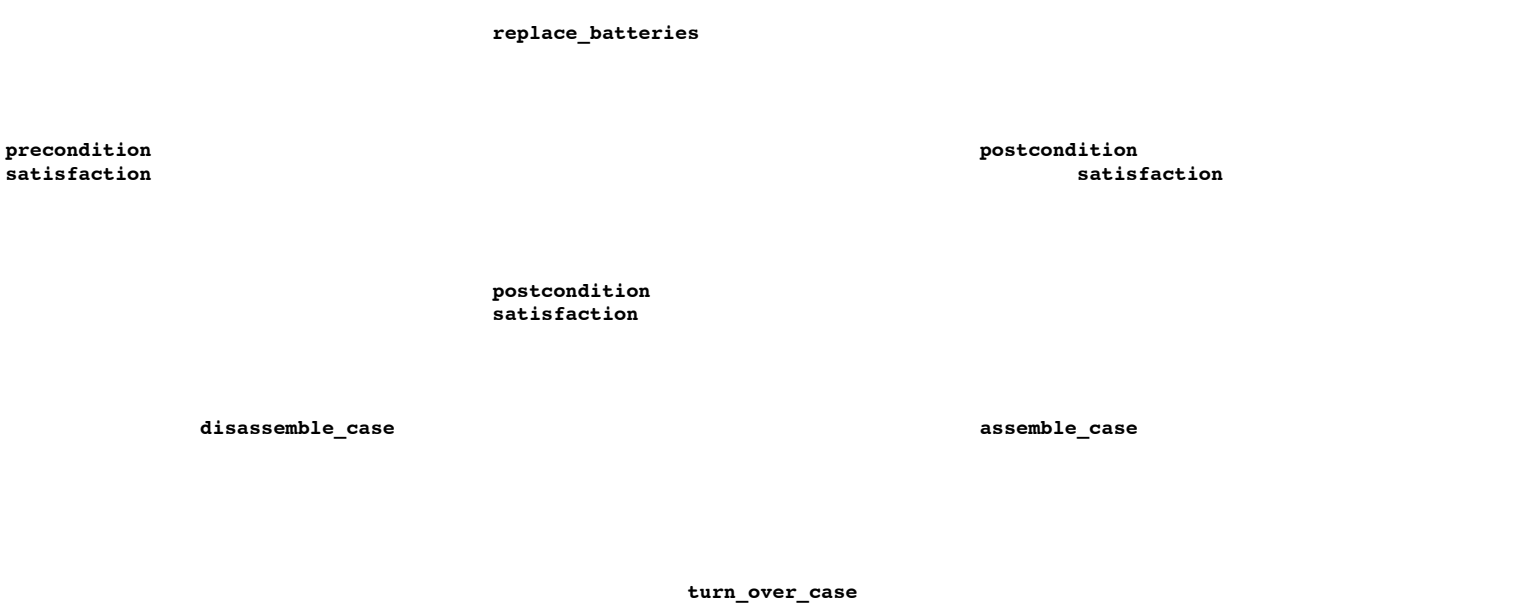


Figure 11-6: the form of the solution of the dead-batteries flashlight problem by means-ends analysis (Figure 11-7 gives a more detailed picture)

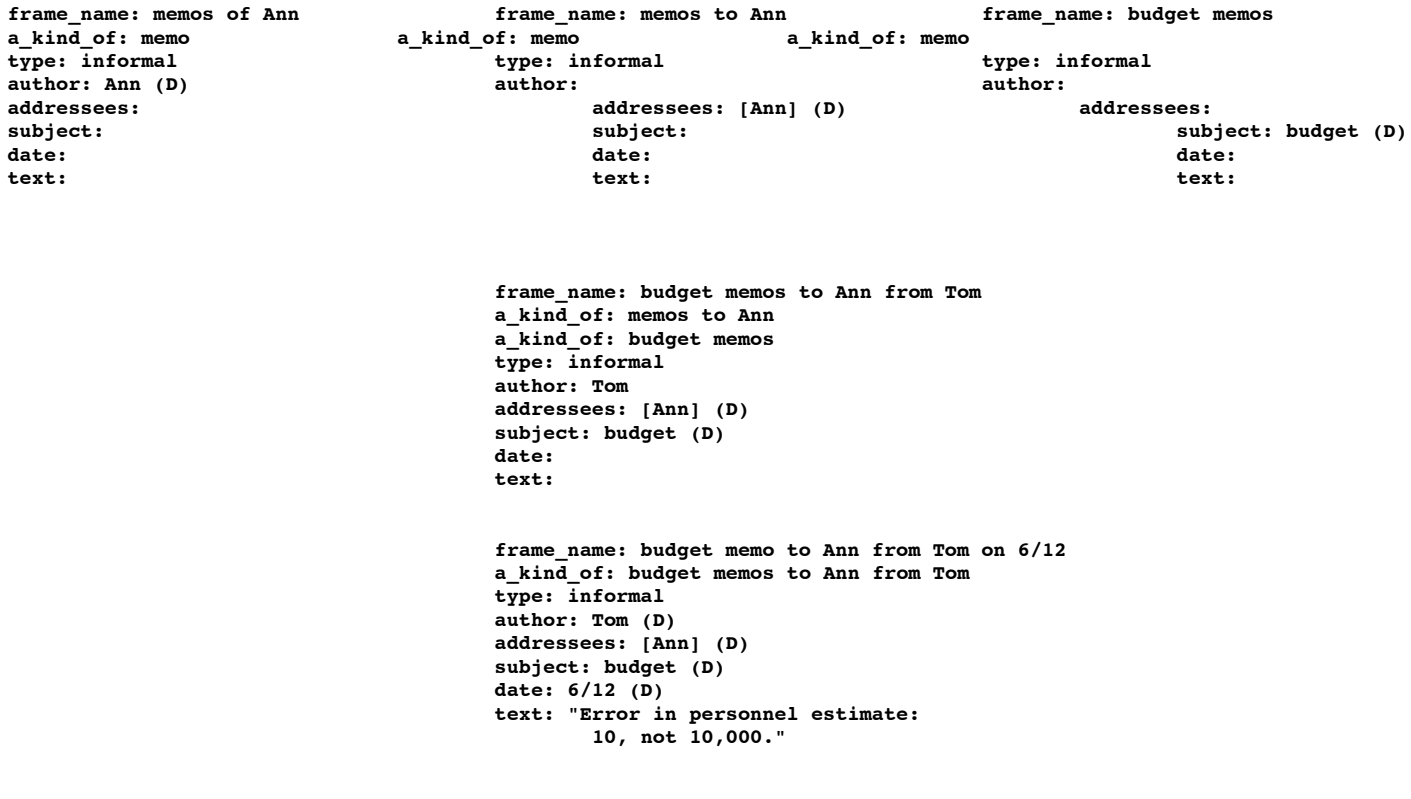
.DS L ***Level 1 State=[closed(case),closed(top),inside(batteries),defective(batteries), ok(light),unbroken(case)] Goal= [ok(batteries),closed(case),closed(top)] Oplist=[disassemble_case,turn_over_case,replace_batteries,assemble_case] Goalstate= [closed(case),inside(batteries),ok(batteries),closed(top), ok(light),unbroken(case)] Operator=replace_batteries Level 2 (Precondition recursion) (Postcondition recursion) State=[closed(case),closed(top), State=[inside(batteries),ok(batteries), inside(batteries),defective(batteries), open(case),closed(top), ok(light),unbroken(case)] ok(light),unbroken(case)] Goal=[open(case),outside(batteries),] Goal=[ok(batteries),closed(case), unbroken(case)] closed(top)] Oplist=[disassemble_case,turn_over_case] Oplist=[assemble_case] Goalstate=[outside(batteries),open(case), Goalstate=[closed(case),inside(batteries), closed(top),defective(batteries), ok(batteries),closed(top), ok(light),unbroken(case)] ok(light),unbroken(case)] Operator=disassemble_case Operator=assemble_case Level 3 (Precondition (Postcondition (Precondition (Postcondition recursion) recursion) recursion) recursion) [nothing State=[open(case),closed(top), [nothing [nothing needed] inside(batteries),defective(batteries) needed] needed] ok(light),unbroken(case)] Goal=[outside(batteries)] Oplist=[turn_over_case] Goalstate=[outside(batteries),open(case), closed(top),defective(batteries), ok(light),unbroken(case)] Operator=turn_over_case Level 4 (Precondition recursion) (Postcondition recursion) [nothing needed] [nothing needed]***

Figure 11-7: the recursive calls in the dead-batteries flashlight problem, with bindings eventually found for Oplist, Goalstate, and Operator

---

```
frame_name: business_form
a_kind_of: physical_object
type:
author:


frame_name: memo
a_kind_of: business_form
type: informal
author:
addressees:
subject:
date:
text:
```



Notes: each box is a frame, and arrows represent a_kind_of facts. Slot names are given, then a colon, and then the slot value; a "(D)" means the value follows by the definition of the frame.

Figure 12-1: some frames about memos

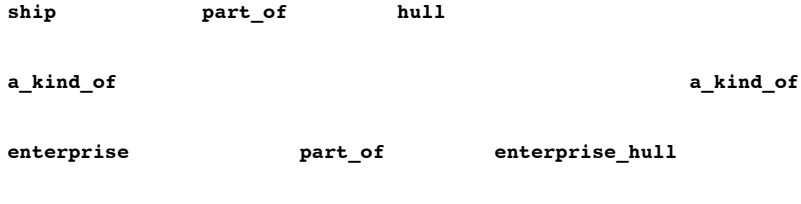
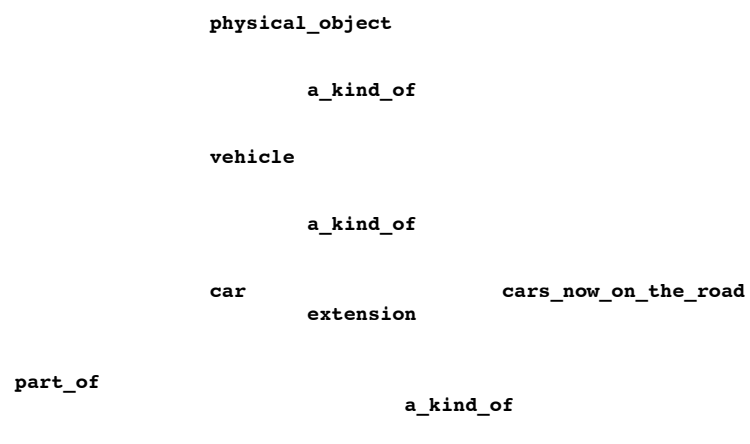


Figure 12-2: an example of part-kind inheritance



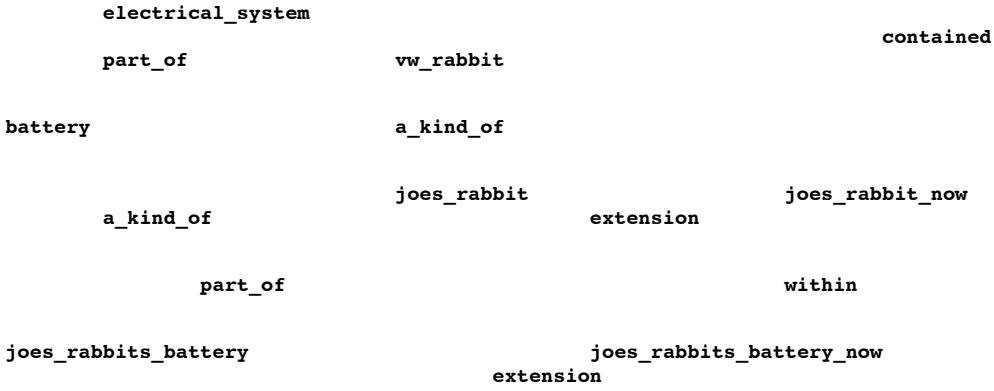


Figure 12-3: the semantic network for the frames in the cars example

---

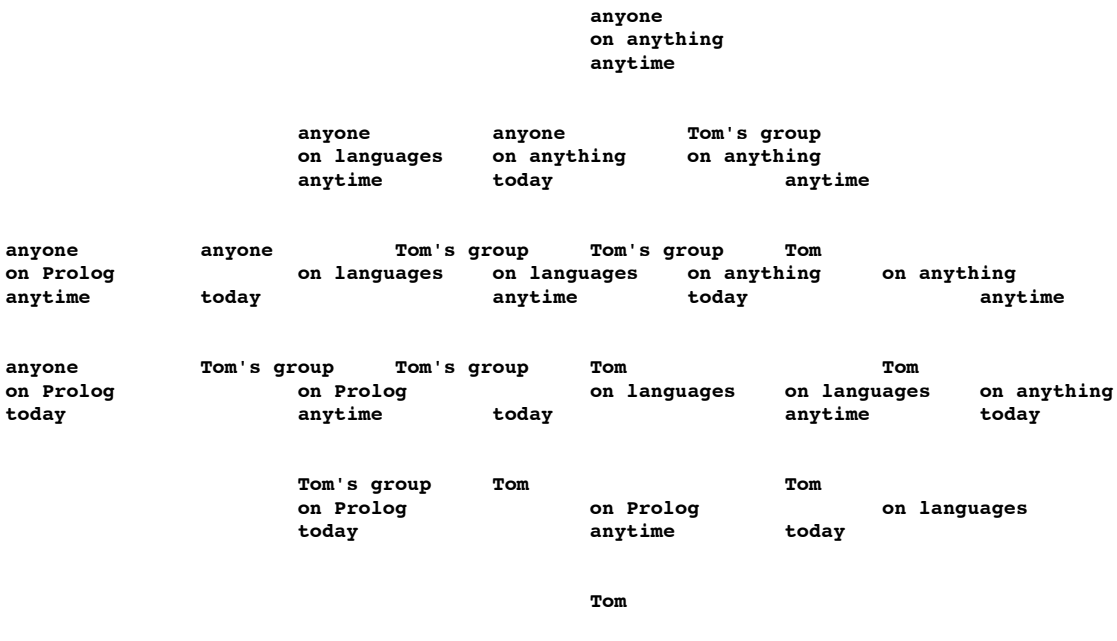
```
frame_name: sending8347
a_kind_of: sending
actor: we
object: memo72185
time: yesterday
destination: headquarters
method: express mail


frame_name: memo72185
a_kind_of: memo
author: Tom
addressees: [Ann]
origin: drawingup20991


frame_name: drawingup20991
a_kind_of: drawingup
actor: Tom
object: memo72185
beneficiary: Ann
time: 6/12
```

.sp 7

Figure 12-4: frame representation of the meaning (semantics) of the sentence "Yesterday we sent headquarters by express mail the budget memo that Tom drew up for Ann on 6/12"



```
on Prolog
today
```

Note: "languages" means "programming languages", "Prolog" means "Prolog interpreter"; all arrows represent a_kind_of

Figure 12-5: a three-dimensional inheritance hierarchy of user model frames

---

```
Time|Monday|Tuesday|Wednesday|Thursday|Friday

9|occupied|occupied|occupied||occupied
10|occupied|occupied|occupied|occupied|occupied
11|class|occupied|occupied|class|occupied
12|occupied|occupied|occupied|occupied|occupied
1|class|class|occupied|class|occupied
2|occupied||occupied||occupied
3|occupied|occupied|occupied|occupied|occupied
4|occupied|occupied|occupied||occupied
```

Figure 13-1: an example scheduling problem, with an example solution symbolized by "class"

```
value|Satisfies|Satisfies|Satisfies|Satisfies|Satisfies
|label?|large?|regular?|irregular?

grass|yes|yes|no|yes
water|yes|yes|no|yes
pavement|yes|yes|yes|no
house|yes|yes|yes|no
vehicle|yes|no|yes|no
```

Figure 13-3: summary of the single-argument predicates in the photo interpretation example

---

```
A1|A2|Satisfies|Satisfies
value|value|borders(A1,A2)?|inside(A1,A2)?

grass|grass|no|no
grass|water|yes|yes
grass|pavement|yes|yes
grass|house|yes|no
grass|vehicle|no|no
water|grass|yes|yes
water|water|no|no
water|pavement|yes|yes
water|house|no|no
water|vehicle|no|no
pavement|grass|yes|no
pavement|water|yes|no
pavement|pavement|no|no
pavement|house|yes|no
pavement|vehicle|yes|no
house|grass|yes|yes
house|water|no|no
house|pavement|yes|yes
house|house|no|no
house|vehicle|no|no
vehicle|grass|no|no
vehicle|water|no|no
vehicle|pavement|yes|yes
vehicle|house|no|no
vehicle|vehicle|no|no
```

Figure 13-4: summary of the two-argument predicates in the photo interpretation example

---

Given the query:

```
?- a(X), b(X,Y), c(Z), d(Y,Z), e(X).
```

These are its dependencies:

Predicate	*Variables*	*Dependencies to*	*Dependencies that bind a common variable*
	expression	*bound*	*previous predicate*
a(X)	X	none	none
b(X,Y)	Y	a(X)	a(X)
c(Z)	Z	none	none
d(Y,Z)	none	b(X,Y),c(Z)	b(X,Y),c(Z)
e(X)	none	a(X),b(X,Y)	a(X)

Figure 13-5: summary of dependencies in an example query

Given the query:

```
?- a(X), b(X,Y), c(Z), d(X,Z), e(X).
```

Given the database:

```
a(1).
a(2).
a(3).
b(A,B) :- B is 3*A.
c(4).
c(1).
d(A,B) :- A>B.
e(3).
```

This is what happens in dependency-based backtracking:

Step	*a(X)*	*b(X,Y)*	*c(Z)*	*d(X,Z)*	*e(X)*
start	active	active	active	active	active
1	X bound to 1	active	active	active	active
2	inactive	Y bound to 3	active	active	active
3	inactive	inactive	Z bound to 4	active	active
4	inactive	inactive	inactive	fails	active
5	active	inactive	Z bound to 1	active	active
6	active	inactive	inactive	fails	active
7	active	inactive	fails	active	active
8	X bound to 2	active	active	active	active
9	inactive	Y bound to 6	active	active	active
10	inactive	inactive	Z bound to 4	active	active
11	inactive	inactive	inactive	fails	active
12	active	inactive	Z bound to 1	active	active
13	active	inactive	inactive	succeeds	active
14	active	inactive	inactive	inactive	fails
15	X bound to 3	active	inactive	active	active
16	inactive	Y bound to 9	inactive	active	active
17	inactive	inactive	inactive	succeeds	active
18	inactive	inactive	inactive	inactive	succeeds

Figure 13-6: a dependency-based backtracking example

---

n	*p*	*Probability*
5	.25	.76
10	.25	.94
20	.25	1.0E+00
50	.25	1.0E+00
5	.1	.4
10	.1	.65
20	.1	.88
50	.1	.99
5	.01	.05
10	.01	.10
20	.01	.18
50	.01	.39
100	.01	.63

Figure 13-7: the probability that some item in a possibility list will be impossible, for an n-item list with probability p that a possibility will be impossible

---

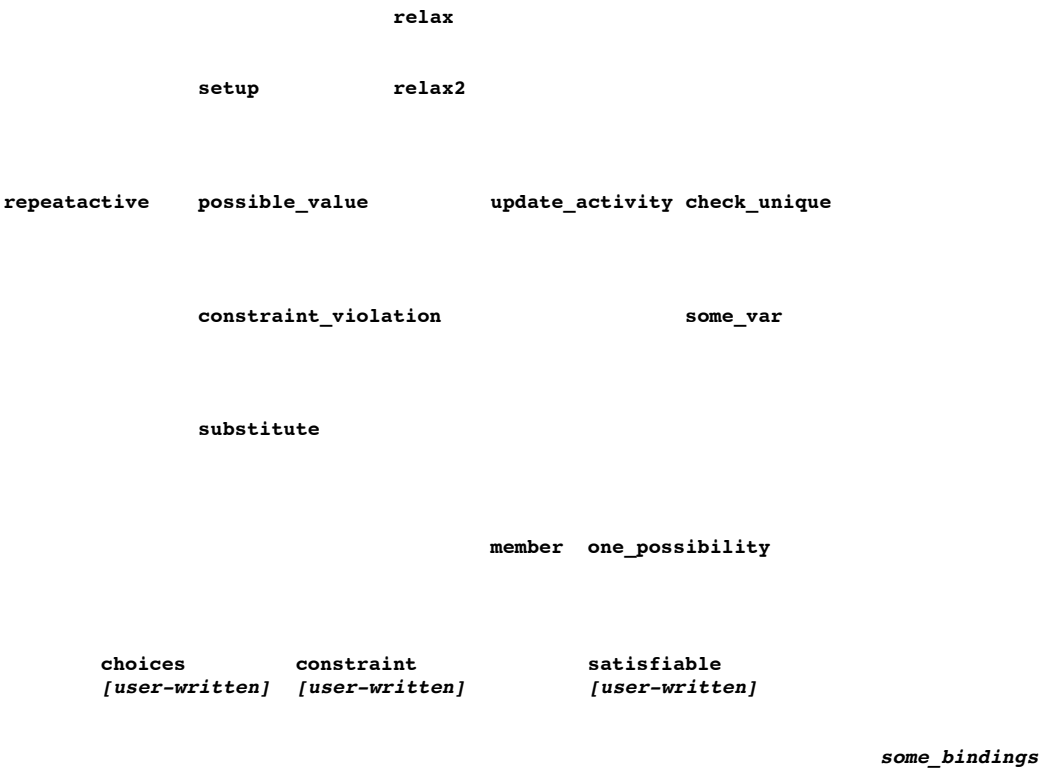


Figure 13-8: the predicate hierarchy for the pure-relaxation program

---

```
a|b|a; not(b)|Is the rule|Justification
|||a :- b. consistent
|||(uncontradicted)?

true|true|true|yes|b proves a


true|false|true|yes|a could be proved
||||by another rule, and
||||that wouldn't violate
||||the truth of this one


false|true|false|no|b proves a by the
||||rule, a contradiction


false|false|true|yes|the rule doesn't
||||apply, and there may
||||be no other rule to
||||prove a
```

Figure 14-1: demonstration that an equivalent logical (declarative) meaning of the Prolog rule a :- b. is a; not(b).

```
not(b); not(c).
[from step 2]

                not(c).
                [from step 3]

                        not(d).
                        [from step 4]

                                "null clause"
                                [from step 5]
```

Figure 14-2: a resolution example

```
In rule form|In clause form

BACKWARDS CHAINING:|
QUERY: ?- a.|ASSUMPTION: not(a).
a :- b, c, d.|a; not(b); not(c); not(d).
RESULTS IN:|RESOLVES TO:
?-b, c, d.|not(b); not(c); not(d).
FORWARDS CHAINING:|
a :- b, c.|a; not(b); not(c).
c.|c.
RESULTS IN:|RESOLVES TO:
a :- b.|a; not(b).
RULE COLLAPSING:|
a :- b, c, d.|a; not(b); not(c); not(d).
c :- e, f.|c; not(e); not(f).
RESULTS IN:|RESOLVES TO:
a :- b, e, f, d.|a; not(b); not(e); not(f); not(d).
```

Figure 14-3: three special cases of resolution

---

1. methods requiring a gold standard

- **(a) traces**
- **(b) confusion matrices**
- **(c) comparison of a near miss and example**
  - **(i) no numbers involved**
  - **(ii) composite results**
  - **(iii) numbers involved**
- **(d) comparison of two examples**
  - **(i) no numbers involved**
  - **(ii) composite results**
  - **(iii) numbers involved**

2. methods not requiring a gold standard

- **(a) schema for rules and facts**
- **(b) user-initiated debugging questions**
- **(c) cooperativeness evaluation**
- **(d) a priori suitability for artificial intelligence techniques**

Figure 15-1: tools for testing and debugging

```
guessed|battery|electrical|fuel|engine
 cause|dead|short|blockage|problem
```

```
actual
cause
battery dead|42|1|4|0
electrical short|15|29|8|3
fuel blockage|0|2|25|1
engine problem|1|0|8|12
```

Figure 15-2: a confusion matrix, for an expert system diagnosing situations in which a car won't start

---

Given the rule a :- b, c.
Assume a, b, c, d, e, f, and g are predicate expressions.
Assume b, c, d, and e are composed of one or more predicate expressions "and"ed and "or"d together.

```
Nature of|Predicate expressions|Suggested new rule
the new case|that are true
|in the new case (those
|not mentioned are false)

example|a and b and c|no change needed
example|a and b|a :- b.
example|a and b and c and d|no change needed
example|a and b and d|a :- b, e.
||where e is implied by both c and d
||(if no such e, try a:- b.)
example|f and b and c|g :- b, c.
||where g is implied by both a and f
||(if no such g, create a new rule f :- b, c.)
near miss|a and b and c|remove rule entirely
near miss|a and b|no change needed
near miss|a and b and c and d|a :- b, c, not(d).
near miss|a and b and d|no change needed
near miss|f and b and c|no change needed
```

Figure 15-3: suggested rule modifications given a new case

```
A|B|A AND B|A OR B|NOT A|

true|true|true|true|false
true|false|false|true|false
false|true|false|true|true
false|false|false|false|true
```

Figure A-1: summary of AND, OR, and NOT

Task: add 1000 to each salary in a set of 732 employee salaries, starting with employee #1

> **Task: add 100 to the salary of employee #1 Task: add 1000 to each salary in a set of 731 employee salaries, starting with employee #2**
>
> > **Task: add 1000 to the salary of employee #2 Task: add 1000 to each salary in a set of 730 employee salaries, starting with employee #3**

Task: add 1000 to each salary in a set of 732 employee salaries, starting with employee #1

> **Task: add 1000 to each salary in a set of 366 employee salaries, starting with employee #1 Task: add 1000 to each salary in a set of 366 employee salaries, starting with employee #367**
>
> > **Task: add 1000 to each salary in a set of 183 employee salaries, starting with employee #1 Task: add 1000 to each salary in a set of 183 employee salaries, starting with employee #184 Task: add 1000 to each salary in a set of 183 employee salaries, starting with employee #367 Task: add 1000 to each salary in a set of 183 employee salaries, starting with employee #550**

Figure B-1: two different ways of doing an example recursion

Task: take the sum of 732 employee salaries, starting with employee #1

(the right box below calculates the result)

> **Task: take the sum of 731 employee salaries, starting with employee #2**
> **Task: add the result in the left box to the salary of employee #1**
>
> **(the right box below calculates the result)**
>
> > **Task: take the sum of 730 employee salaries, starting with employee #3**
> > **Task: add the result in the left box to the salary of employee #2**

Figure B-2: an example of recursion of a function

---

array: list: stack using an array (insert and delete only at right end): queue using an array (insert at left end, delete at right end): tree: .sp 8

lattice: .sp 8

graph: .sp 8

Figure C-1: pictures of examples of some famous data structures

---

```
Built-in|Arguments|Can it succeed on|Description
predicate||backtracking?

consult|a filename|no|loads a file
listing|a predicate name|no|prints database items
|||with that predicate name
asserta|a fact|no|adds a fact to the front
|||of the database
assertz|a fact|no|adds a fact to the rear
|||of the database
retract|a fact|no|removes a fact
abolish|a predicate name|no|removes all facts with
|and its number||the same predicate name
|of arguments|
> [infix]|two numbers|no|greater-than
< [infix]||two numbers|no|less-than
= [infix]|two numbers|yes|equals
is [infix]|variable and|yes|arithmetic assignment
|a numeric expression|
number|a variable|no|succeeds if argument
|||is bound to a number
write|a variable or|no|prints on the terminal
|character string
read|a variable or|no|reads a word typed
|character string||by the user
nl|none|no|issues carriage return to
|||the terminal
not|a predicate|no|succeeds if querying its
|expression||argument fails
fail|none|no|always fails
!|none|no|[see text]
var|a variable|no|succeeds if argument
|||is not bound to anything
call|a predicate|yes|queries the predicate
|expression
clause|a rule left side|yes|binds text of a rule
|and rule right side||to any variables
=.. [infix]|a query|yes|converts a query to a
|and a list||list or vice versa
```

Figure D-1: summary of built-in Prolog predicates assumed in this book

---

```
Standard Prolog|Micro-Prolog
feature|feature

consult(<filename>)|LOAD <filename>
listing(<predicatename>)|LIST <predicatename>
```

```
asserta(F)|(ADDCL F 0)
assertz(F)|(ADDCL F <number-of-F-facts-and-rules>)
retract(F)|(DELCL F)
abolish(F,2)|KILL F
?- |&.
:-|[see discussion]
,|[see discussion]
p(X); q(X)|(OR ((p X)) ((q X)))
not(p(X))|(NOT p X)
X<Y|(LESS X Y)
X>Y|(LESS Y X)
X=3|(EQ X 3)
X is 3|(EQ X 3)
X is Y+Z|(SUM Y Z X)
X is Y-Z|(SUM X Z Y)
X is Y*Z|(TIMES Y Z X)
X is Y/Z|(TIMES X Z Y)
[a,b,X]|(a b X)
[X|Y]|(X | Y)
'Character string'|(Character string)
write(X)|(P X)
write(X),nl|(PP X)
read(X)|(R X)
clause(p(X),Y)|((p X) | Y)
P=..L|[unnecessary--queries are lists]
call(p(X))|(? ((p X))
fail|(FAIL)
!|!
number(X)|(NUM X)
var(X)|(VAR X)
/* comment */|/* comment
setof(X,p(X),X2)|(ISALL X2 X (p X))
```

Figure E-1: approximate equivalents between the Prolog assumed in this book and Micro-Prolog

---

```
                      a

        b             c             d

e       f       g     h
```

Figure G-1: answer to problem 9-11.(a)

Go to paper index