Prolog: extending 'Pure' Prolog'

Prolog has some **in-built** predicates (**'primitives'**), e.g. `fail, retract, findall, is`

A goal containing such a predicate is not solved by resolution (Prolog doesn't look for matching facts/rules). They are **extra-logical**.

Instead, a procedure (e.g. written in C) is associated with these predicates: **procedural attachment**.

And this procedure is run: the predicates are **evaluable**.

These extensions to 'pure' Prolog are not needed. ('Pure' Prolog offers all the power of a Turing machine). But they can give us less cumbersome and more efficient programs.

Their disadvantages are that they are rarely polymodal (getting instantiation patterns right is crucial) and they make us think about ordering even more.

Negation

Prolog does not need a form of negation.

```
mother(chas, liz).      mother(andy, liz).
father(chas, phil).     father(andy, phil).

sibling(X, Y) :-
   mother(X, M), mother(Y, M),
   father(X, F), father(Y, F),
   different(X, Y).

different(chas, liz).
different(chas, phil).
different(chas, andy).
etc.

neq(0, s(X)).
neq(s(X), 0).
neq(s(X), s(Y)) :- neq(X, Y).
```

But both are cumbersome and the second is inefficient.

Negation as Failure

Prolog does provide a form of negation, `\+`

It is not logical negation; it is extra-logical, defined by procedural attachment, which reduces polymodality and declarativeness. It is useful for giving less cumbersome, more efficient definitions.

A goal `\+P` will be assumed to be true if `P` cannot be proved.

Negation examples

```
brassica(sprouts).
```

```
?- brassica(sprouts).
yes
?- \+ brassica(sprouts).
no
?- brassica(kohlrabi).
no
?- \+ brassica(kohlrabi).
yes
```

```
healthy(X) :- \+ saturated_fat(X).
```

```
?- healthy(lard).
yes
```

```
saturated_fat(lard).
```

```
?- healthy(lard).
no
```

Negation as Finite Failure

- We have seen you can put `\+` into a query or the body of a rule.

- You cannot put `\+` into the head of a rule or in a fact:
  — you cannot *store* that something isn't provable.

- Negation as failure is *really* negation as *finite* failure (NFF):

  > `\+ P` succeeds if `P` cannot be finitely proved (i.e. if `P` finitely fails).

  (Trying to prove `P` might not terminate.)

Closed World Assumption

- Think of NFF as an additional inference rule. Then it needs a semantics.

- Closed world assumption (CWA):
  - everything that is true is in the program or can be derived from it
  - everything that is not known to be true must be false (we need not say something that is false).

- NFF is then a procedural interpretation of the CWA:
  - everything that is not provable is assumed to be false.

Closed World Assumption: Example

```
nut(almond).
nut(cashew).
nut(pecan).
```

```
    ?- nut(pistachio).
    no
    ?- \+ nut(pistachio).
    yes
```

You must ensure that

- all true facts *are* recorded, or

- you have an appropriate understanding of Prolog's answers (you could even change the names of your predicates symbols to remind you of this, e.g. `nutsPrologKnows`).

Prolog's negation is not sound

- A negated Prolog goal may not produce what it ought if the goal contains unbound variables when it is called.

- Example

```
student(barry).
student(gary).
married(barry).
```

— the unmarried students: `gary`

```
?- student(X), \+ married(X).
X = gary ;
no
?- \+ married(X), student(X).
no
```

Unsound negation: explanation

- When a goal fails, Prolog backtracks, undoing substitutions.

- For `\+ P` to succeed,
  `P` must fail,
  but if `P` fails,
  there won't be any substitutions.

- So
  - substitutions can be communicated *into* negated goals
  - but they cannot be communicated *out* of them,

  which makes subgoal ordering more important.

Failure-Driven Loops

Prolog's negation can be used to force backtracking.

```
qual(pers1, typing5). entry(job12, piano4).
qual(pers1, frenchA). entry(job12, frenchA).
qual(pers1, piano4).  entry(job17, csBsc).
qual(pers2, csBsc).

shouldApply(P, J) :-
   \+ (entry(J, Q),
       \+ qual(P, Q)).
```

```
?- shouldApply(pers1, job12).
yes
?- shouldApply(pers2, job12).
no
?- shouldApply(pers2, job34).
```

yes !!! — we ought to make sure the job exists

Failure-Driven Loops: another example

```
cleverSubset(Xs, Ys) :-
    \+ (member(Z, Xs),
        \+ member(Z, Ys)).
```

```
?- cleverSubset([a, b], [c, a, b, d]).
yes
?- cleverSubset([a, b], [a, c, d]).
no
```

Warnings

If `member` not loaded:

```
?- cleverSubset([c, d], [a, b, c])
yes
```

If called inappropriately:

```
?- cleverSubset(a, [a, b, c]).
yes
```

Some in-built predicates

- `retract(G)`
  — deletes from the database a clause whose head matches `G`.

- `fail`
  — a goal that always fails. (Any predicate that has no definition in the database could be used instead.)

- `true`
  — a goal that always succeeds.

Suppose we want a program to delete all `parent/2` facts from the Prolog database.

```
parent(liz, chas).
parent(chas, willie).
parent(di, willie).
```

Version 1: a recursive call

```
kill_parent :- retract(parent(X, Y)),
               kill_parent.

kill_parent.
```

— use of Prolog impurities (`retract`) has again forced us to re-think our ideas about ordering. The more usual ordering would not work:

```
kill_parent.

kill_parent :- retract(parent(X, Y)),
               kill_parent.
```

Version 2: a failure-driven loop that uses `fail`

```
kill_parent :- retract(parent(X, Y)),
               fail.

kill_parent.
```

—the second clause is optional. (Question: what's its point?) The ordering is significant: the following doesn't work:

```
kill_parent.

kill_parent :- retract(parent(X, Y)),
               fail.
```

The failure-driven version is more (space)-efficient than the recursive version.

Version 3: a failure-driven loop that uses `\+`

```
kill_parent :-
   \+ (retract(parent(X, Y)),
       \+ true).
```

— this works in basically the same way as version 2 (by forcing failure and hence forcing backtracking), but encapsulates the definition into a single clause.

These programs had a side-effect (deletion from the database), and we saw that the failure-driven versions were more efficient than the recursive version.

We'll now look at more typical programs, without side-effects, and ask: when should we use failure-driven loops and when should we use recursion? (The answer is no longer a matter of efficiency.)

Going loopy

```
age(sid, 59).          age(vera, 64).
age(alf, 92).          age(betty, 256).

oap(P) :- age(P, A), A > 60.
```

Define a predicate `all_oaps/1`, where `all_oaps(L)` is true if list `L` contains only O.A.P.s, e.g.:

```
?- all_oaps([vera, alf]).
yes
```

```
all_oaps([]).

all_oaps([P | Ps]) :- oap(P), all_oaps(Ps).
```

— a recursive data structure, so use a recursive definition. (Trees, expressions of arithmetic or logic, Peano numbers, etc. are all recursive data structures and should be processed using recursive definitions.)

Same problem again

Suppose the people are not given to us in a list. Instead they are given as facts in the database:

```
person(vera).
person(alf).
person(betty).
```

Define a predicate `all_oaps/0`, where `all_oaps` is true if all the `person`s in the database are O.A.P.s, e.g.:

```
?- all_oaps.
yes
```

Solution

This cannot be done recursively: facts in the database do not form a recursive data structure. To ‘loop’, we must force backtracking.

```
all_oaps :-
   \+(person(P),
      \+ oap(P)).
```

or

```
all_oaps :- person(P), oap(P), fail.

all_oaps.
```

Collecting things when going loopy

Define a predicate `ages/2`, where `ages(L, A)` is true if `A` lists the ages of each of the people in list `L`, e.g.

```
?- ages([vera, alf, betty], As).
As = [64, 92, 256]
```

```
ages([], []).

ages([P | Ps], [A | As]) :-
    age(P, A), ages(Ps, As).
```

Same problem again

Suppose again the people are not in a list, so we cannot use recursion.

Can we use these failure-driven loops?

```
ages(As) :-
    \+(person(P),
        \+ age(P, A)).
```

or

```
ages(As) :- person(P), age(P, A), fail.

ages(As).
```

No. Neither of these works: when we backtrack, variable instantiations are undone, so we lose the ages we are trying to collect.

All-solutions predicates

- These are in-built predicates
  - useful for processing non-recursive data structures (deep down they're failure-driven loops)
  - but unlike other failure-driven loops, they collect up substitutions.

- `findall(Term, Goal, Bag)`
  — `Bag` is the list of all instances of `Term` such that `Goal` is provable.

Daddy?

```
father(phil, chas).    father(chas, willie).
father(phil, andy).    father(chas, henry).
father(phil, eddie).
```

```
?- findall(X, father(phil, X), Xs).

?- findall(X, father(X, chas), Xs).

?- findall(X, father(eddie, X), Xs).

?- findall(pair(X, Y), father(X, Y),
                                XYs).



?- findall(Y, father(X, Y), Ys).
```

Old people again

We return to defining `ages/1`, where `ages(A)` is true if `A` lists the ages of `person`s in the database:

```
ages(As) :-
    findall(A, (person(P), age(P, A)), As).
```

Alternatively, since `findall/3` is good for converting non-recursive data structures into recursive ones, we could collect the people into a list and then use the recursive definition of `ages/2` on that list:

```
ages(As) :-
    findall(P, person(P), Ps), ages(Ps, As).
```

Arithmetic

- Logic programs compute truth, not values.

- But logical reasoning may need to be quantitative.

- Arithmetic on natural numbers can be axiomatised (Peano's axioms):
  — elegant, but inefficient (e.g. the cost of recursion in `add/3`), fails to exploit machine's capabilities.

- Logic programming languages are extended with extralogical arithmetic:
  built-in/ procedural attachment/ evaluable/ primitives

Should the arithmetic operators be function symbols or predicate symbols?

Why arithmetic operators should be predicates

- `append`, which can be thought of as addition over lists, is defined as a 3-ary predicate. Therefore, `add` and other operators over numbers should be similar.

- Addition and subtraction are modal variants of the same relation. If addition is provided as a 3-ary predicate, it could be used to achieve

  ```
  X + Y => Z; Z - X => Y; Z - Y => X
  ```

  (Similarly multiplication and division.) Hence, the advantage of polymodality.

- Function symbol arithmetic could be implemented by the programmer in terms of the predicates if s/he so desired.

Why arithmetic operators should be functions

- Composition of arithmetic operators is desirable, and only functions compose, e.g.

functions	`Answ = B * B - 4 * A * C`

predicates	`mult(B, B, Temp1),`
	`mult(4, A, Temp2),`
	`mult(Temp2, C, Temp3),`
	`sub(Temp1, Temp3, Answ)`

- There seems to be an obvious relationship of *evaluation* between an arithmetic *term* and its *value*.

- Predicate symbol arithmetic could be implemented by the programmer in terms of the functions if s/he so desired.

Using predicates would be the logic programming idea, but ‘standard’ Prologs offer their arithmetic through function symbols.

Prolog arithmetic

- Function symbols are not normally evaluated in Prolog (even `+`, `-` `*`, `/`), e.g.:

```
derivative(F + G, DF + DG) :-
 derivative(F, DF), derivative(G, DG).
```

```
  ?-derivative(x↑s(0)+x↑s(0), X).
  X = s(0)*x↑0+s(0)*x↑0
```

- Now that certain function symbols are evaluable, we signal to Prolog when we wish the function symbols to be evaluated by using them in certain syntactic contexts:

  – `Value is Expression`

  – comparison operators.

Value is Expression

- `is` is an in-built, *infix, binary predicate*.

- For a goal of the form

```
Value is Expression
```

  - `Expression` is evaluated as an arithmetic expression (if possible)

  - the result of a successful evaluation is *unified* with `Value`

  - if the unification succeeds, the goal as a whole succeeds; else the goal fails.

Value is Expression: examples

```
?- 8 is 3 + 5.
```

```
?- X is 3 + 5.
```

```
?- 9 is 3 + 5.
```

```
?- 3 + 5 is 3 + 5
```

Value is Expression: not polymodal

- Since arithmetic is done using functions, it is no surprise that we don't have polymodality.
- For example, if the database contains

```
prime(5).
```

```
?- 8 is X + 3.
```

```
?- prime(X), 8 is X + 3.
```

```
?- X is X + 1.
```

```
?- prime(X), X is X + 1.
```

Value is Expression: mode restriction

- Mode restriction: Prolog can solve a goal `Value is Expression` only if `Expression` is fully instantiated when it is called.

- Example: `sumlist(L, N)` is true if the sum of the numbers in list `L` is equal to `N`

```
?- sumlist([1, 2, 3], X).
X = 6
```

- Incorrect

```
sumlist([], 0).

sumlist([N | Ns], S) :-
   S is N + SNs, sumlist(Ns, SNs).
```

— this doesn't have any modes at all.

Value is Expression: mode restriction cont'd.

- Correct

```
sumlist([], 0).

sumlist([N | Ns], S) :-
    sumlist(Ns, SNs), S is N + SNs.
```

  - left-recursion forced onto us
  - not polymodal: first argument must be fully instantiated.

Numeric comparison operators

- $<, >, =<, >=, =:=, = \backslash =$ are in-built *infix binary predicates*

- In, e.g., `Expr1 < Expr2`, the two expressions are evaluated as arithmetic expressions and the results are compared.

- If one or both of `Expr1` or `Expr2` is not an arithmetic expression or is not fully instantiated when evaluated, the goal raises an error condition (hence, again, a loss of polymodality).

Comparison operators: examples

- E.g. if the database contains `prime(5).`

```
?- 3 - 2 < 2 * 3 + 1.

?- 2 < 1.

?- clyde < 3.

?- X < 1.

?- prime(X), X < 1.
```

What does = equal?

- Prolog has definitions of `=`, `=:=` and `is` built-in. What's the difference?
  - `Expr1 = Expr2`: simply tests for unification; neither expression is evaluated.
  - `Expr1 is Expr2`: causes `Expr2` to be evaluated as an arithmetic expression (if possible), and the result is unified with `Expr1`. (Since the result will be some number, `Expr1` will need to be a variable or a variable if the unification is to succeed.)
  - `Expr1 =:= Expr2`: causes both `Expr1` and `Expr2` to be evaluated as arithmetic expressions (if possible) and the results are tested for equality (i.e. the results should be equal numbers if this is to succeed).

Greatest Common Divisors

- GCD(x, y):
  - if $x = y$, then GCD(x, y) = x
  - if $x < y$, then GCD(x, y) = GCD($x, y - x$)
  - if $y < x$, then GCD(x, y) = GCD($y, x - y$)

```
gcd(X, X, X).

gcd(X, Y, D) :-
   X < Y,
   Y1 is Y - X,
   gcd(X, Y1, D).

gcd(X, Y, D) :-
   Y < X,
   X1 is X - Y,
   gcd(X1, Y, D).
```

Primes

```
primes(Limit, Ps) :-
    integers(2, Limit, Is), sift(Is, Ps).

integers(Low, High, [Low | Rest]) :-
    Low =< High,
    M is Low + 1,
    integers(M, High, Rest).

integers(Low, High, []) :-
    Low > High.

sift([], []).

sift([I | Is], [I | Ps]) :-
    remove(I, Is, New),
    sift(New, Ps).
```

```
remove(P, [], []).

remove(P, [I | Is], [I | Nis]) :-
    \+ (0 is I mod P),
    remove(P, Is, Nis).

remove(P, [I | Is], Nis) :-
    0 is I mod P,
    remove(P, Is, Nis).
```

Controlling Backtracking

- Prolog is ideal for 'non-deterministic' programs — e.g. permutation sort, A.I. planners, list membership, deletion from a list

```
del(X, [X | Xs], Xs).

del(X, [Y | Ys], [Y | Zs]) :-
    del(X, Ys, Zs).
```

- But some programs are deterministic (e.g. they compute functions — one answer)
  — in these cases, on backtracking after finding the solution, Prolog explores fruitless branches: harmless but inefficient.

The cut

- Prolog provides the cut operator (`!`)
  - so programmers can prune paths s/he knows cannot produce solutions
  - inappropriately, s/he can also prune paths that do contain solutions
  - inappropriately, it leads some programmers to write declaratively incorrect programs
- The cut
  - has no declarative semantics
  - is an operator that affects the behaviour (procedural semantics)

Preventing backtracking

- Suppose you have a goal `P` which matches the head of all of these:

```
P1 :- P11, ..., P1n.          (1)
P2 :- P21, ..., P2i, !,       (2)
             P2j, ..., P2k.
P3 :- P31, ..., P3p.          (3)
```

  - (1) is executed normally.
  - In (2), on reaching the cut, Prolog commits to all choices made since `P` was unified with P_2,
    - solutions to P_{21}, ...P_{2i} are frozen
    - P_{2j}, ..., P_{2k} can be solved in as many ways as possible
    - does not use (3)

- NO BACKTRACK PRIOR TO CUT

Exercise in cutting

What answers do we get if this program

```
p(X, Y) :- q(X, Y).
p(a, b).
q(c, d).
q(e, f).
q(X, Y) :- r(X), !, s(Y).
q(X, Y) :- t(X, Y).
r(e).
r(f).
s(g).
s(h).
t(i, j).
```

is run for the following goal

```
?- p(X, Y).
```

assuming the user types `;` at every opportunity?

```
?- p(X, Y).
```

```
p(X, Y) :- q(X, Y).

p(a, b).

q(c, d).

q(e, f).

q(X, Y) :- r(X), !, s(Y).

q(X, Y) :- t(X, Y).

r(e).

r(f).

s(g).

s(h).

t(i, j).
```

Green cuts

- The program for computing the minimum of two numbers is really deterministic. We can actually make it so by inserting a cut so that Prolog can dynamically prune further fruitless search.

```
min(X, Y, X) :- X =< Y, !.

min(X, Y, Y) :- X > Y.
```

- Green cut
  - the program is more efficient
  - but declarative meaning is unaffected
    * can remove the cut with impunity
    * the program isn't more 'order dependent'
    * can understand the program, ignoring the cut

Red cuts

- `member` was a non-deterministic program (consider mode `member(-, ++)`). A cut can make it deterministic:

```
member(X, [X | Xs]) :- !.

member(X, [Y | Ys]) :- member(X, Ys).
```

  - this cut prunes solutions
    (it is fine for mode `member(++, ++)` but for other modes, e.g. `member(-, ++)`, solutions are lost)
  - it makes the declarative and procedural meanings diverge
  - this cut is red, but we might be more detailed and say that it is green w.r.t. mode `member(++, ++)` but red for all other modes

Deterministic member

- If you really want a deterministic `member`,
  - write a declaratively correct deterministic `member`

```
member(X, [X | Xs]).

member(X, [Y | Ys]) :-
    \+ X = Y, member(X, Ys).
```

  - can then improve the efficiency of this program with a green cut

```
member(X, [X | Xs]) :- !.

member(X, [Y | Ys]) :-
    \+ X = Y, member(X, Ys).
```

  - however, no-one is ‘fussy’ enough to ever bother to do things so properly!

Cut considered harmful

- Cuts lead programmers to make more radical (but incorrect) changes to programs.

  ```
  min(X, Y, X) :- X =< Y, !.
  min(X, Y, Y).
  ```

  – even more efficient, but

  – exploits Prolog's sequential processing of rules

  – cannot understand it without understanding the cut (e.g. without the cut, multiple answers for `?- min(2, 3, M).`)

  – the original `min` program (and the version with the green cut) provided two modes, but the program above is correct for only one mode (`min(++, ++, -)`), but not for the other mode (`min(++, ++, ++)`)

    ```
    ?- min(1, 2, 2).
    ```

Cutting exercise

```
insertsort([], []).

insertsort([X | Xs], SortedList) :-
 insertsort(Xs, SortedTail),
 rightful_insert(X, SortedTail, SortedList).

rightful_insert(X, [], []).

rightful_insert(X, [Y|Ys], [X,Y|Ys]) :-
   X =< Y.

rightful_insert(X, [Y|Ys], [Y|Zs]) :-
   X > Y, rightful_insert(X, Ys, Zs).
```

Questions

Consider only the 'typical' modes `insertsort(++, −)` and `rightful_insert(++, ++, −)`.

A cut won't help improve the efficiency of `insertsort/2`. Why not?

A cut can improve the efficiency of `rightful_insert/3`. Where should it go?

Is your cut red or green?

Change its colour.

Making cuts

```
censored(bother).
censored(drat).
etc.

beep([], []).

beep([W | Ws], [beep | Cs]) :- !,
   censored(W), beep(W, Cs).

beep([W | Ws], [W | Cs]) :-
   beep(Ws, Cs).
```

Comment on this cut.

Metaprogramming

- Metaprograms treat other programs as data e.g.
  - compilers
  - interpreters
  - editors
  - tracers
  - debuggers

  also
  - A.I. planners
  - parsers
  - expert system shells

Metaprogramming

- Prolog makes metaprogramming easy
  - can even write Prolog interpreters/ compilers in Prolog
  - Prolog *terms* can encode Prolog *clauses*
    e.g.
    `king(egfrith).`
    — a fact: `king` is the predicate
    `monarch(king(egfrith)).`
    — a fact: `king` is a function symbol
  - writing our own interpreters allows us to
    - add new constructs
    - implement new evaluation strategies

Some evaluable predicates

`read(X)`	read a term from input device and unify with `X`
`write(X)`	term `X` is written to output device
`nl`	newline is written to output device
`tab(N)`	`N` spaces are written to output device

Using the predicates

```
pointless :- write('What? '),
             read(X),
             write(X),
             write(' is rubbish'),
             nl,
             pointless.
```

```
?- pointless.
What?  egfrith.
egfrith is rubbish
What?  king(wulfhere).
king(wulfhere) is rubbish
What?
```

More evaluable predicates

`retract(X)`	already discussed
`assert(X)`	if possible, term `X` is added to the Prolog database as a new clause

E.g.

```
?- assert(father(ethelwulf, ethelbald)).
yes
?- assert((son(X, Y) :- father(Y, X))).
yes
?- son(ethelbald, ethelwulf).
yes
?- retract(son(A, B)).
yes
?- son(ethelbald, ethelwulf).
no
```

Yet more evaluable predicates

`call(X)` if possible, term `X` is executed as if that term appeared textually in place of `call(X)`

E.g. in the database

```
monarch(king(pendu)).
```

`?- call(monarch(king(pendu))).`
— same as `?- monarch(king(pendu)).`

`?- monarch(X), call(X).`
— same as `?- king(pendu).`

`?- call(3).`
— error

`?- call(X), monarch(X).`
— error

Even more evaluable predicates

`clause(P, Q)` by the time this is executed, `P` must be bound to a non-variable. It searches the database for a clause (fact or rule) whose head matches `P`. `Q` will then be bound to the rule's body or to `true` for facts.

```
p(a).
p(X) :- q(X).
p(X) :- r(X), s(X).
```

```
?- clause(p(Z), Q).
Q = true
Z = a ;
Q = q(X)
Z = X ;
Q = r(X), s(X)
Z = X ;
no
?- clause(P, Q).
```

— error

Prolog interpreters: A one-clause interpreter

```
solve(Goal) :- call(Goal).
```

```
?- solve(member(X, [a, b, c])).
X = a ;
```

etc.

- In the above, *all* work is delegated to the underlying Prolog interpreter
  - largely pointless
  - could use it within an interpreter that modifies the Prolog prompt, logs interaction to a file, etc.

```
dialogue :- query, dialogue.

query :- nl, write('Yes? '), read(Goal),
     solve(Goal), write(Goal), nl, fail.

query.
```

Prolog interpreters: A three-clause interpreter

- This interpreter faithfully mimics the operation of the normal Prolog interpreter
  - for pure Prolog
  - uses `clause`

```
solve(true).

solve(Goal) :-
    clause(Goal, Body),
    solve(Body).

solve((Body1, Body2)) :-
    solve(Body1),
    solve(Body2).
```

Absorption

- What does the underlying Prolog interpreter do?

  - in the 1-clause interpreter, *all* the work is absorbed

  - in the case of the 3-clause interpreter,

Absorbed:	unification of goals within clause heads; order in which clauses are tried.
Not absorbed:	choice of subgoals.

- That which *isn't* absorbed is open to modification. Absorb less, modify more.
  — in the 3-clause interpreter can select rightmost literal first
  — in other interpreters, could modify unification/ search

Expert Systems: What are they?

- Expert systems:
  - interpret, diagnose, predict, instruct, monitor, analyse, consult, plan, design
  - in domains requiring human expertise
- An A.I. success story?
  - R1 (XCON) checks customer orders for DEC
  - Personal Financial Planning System gives financial advice
  - PROSPECTOR evaluates geological sites for likely yield of valuable deposits.

Expert Systems: Why build them?

- Advantages
  - disseminate rare and costly expertise
  - formalise expert knowledge
  - integrate diverse sources of knowledge
  - fairness? even-handedness? dispassionate? untiring?
- Disadvantages
  - lack of insight, compassion, commonsense, etc.
- Must consider the ethical issues
  - use them as assistants to experts, not replacements

Rule-Based Systems

- The *most common* way to implement expert systems.

- The **knowledge base** contains
  - **rules**, capturing general knowledge about the domain
  - **facts**, pertaining to the particular case currently under investigation.

- The **inference engine** uses facts, elicited from the use through the **user interface**, to instantiate rules, thus establishing the conclusions of the rules.

A Rule-Based System

One way to represent rules in Prolog

```
rule(goto_heaven(X), [sinned(X),
                      repented(X)]).
rule(goto_hell(X), [sinned(X),
                    dont_care(X)]).
rule(goto_line_100(X),
             [programs_in_BASIC(X)]).
rule(sinned(X), [slothful(X)]).
rule(sinned(X), [tory(X)]).
rule(slothful(X),
          [done_<_16_worksheets(X)]).
rule(slothful(X), [sleeps_til_10(X),
                   then_sleeps_til_3(X)]).
```

As they are collected, facts are represented thus:

```
true(tory(genghis)).
untrue(repented(genghis)).
```

Explanations

- Explanations are important for user confidence in the system and debugging.
- Rule-based expert systems typically provide two types of explanation
  - WHY
    * when the system asks the user a question
    * “Why are you asking me this question?”
    * rules that could fire
  - HOW
    * once system has reached a conclusion
    * “How did you come to that conclusion?”
    * rules that did fire