Sequent Users' Resource Forum

Newport Beach, CA

September 1988

Parallel Programming in Prolog

Robert M. Keller

Quintus Computer Systems, Inc
Mountain View
California

What Prolog is:

A major landmark in language design, comparable in scope to the invention of Fortan, Algol, Lisp, Cobol.

An language usually implemented with an interactive environment.

A language in which complete applications can be programmed.

A language within several built-in features not found in other languages, including:

- Pattern-matching via unification
- Backtracking
- Integral database
- Comprehensive meta-facilities

A language which strongly relates to certain aspects of logic.

A language in which relational or entity-relation database concepts are naturally expressed.

A language with only a very modular syntax and only 2-levels of lexical scoping.

What kinds of applications are suitable for Prolog?

Database applications

Compilers

Rapid Prototyping of all kinds (including AI)

Expert systems, and other applications requiring meta facilities

Program-transforming programs

Prolog Users

- Universities
- Research institutions
- Government agencies
- Corporate
  - AI groups
  - Research and development
  - MIS
- System integrators / application developers

Prolog Application Areas

- Knowledge based systems
  - Fault analysis
  - Diagnosis
  - Configuration
  - Monitoring complex situations
- Components of traditional applications
  - Design
  - Intelligent front ends
  - Compilers, generators
  - Translators

Some Prolog Application Markets

- Manufacturing (aerospace, automobile, electronics)

- CAD (electronic, mechanical, architectural)

- Database, decision support (financial)

- CASE (software engineering)

Some History of Prolog

1965	J.A. Robinson	Syracuse	Development of Resolution principle
1973	R. Kowalski	Edinburgh	Predicate Logic as a Programming Language
1975	A. Colmerauer	Marseille	Metamorphosis-Grammar Language
1975	P. Roussel	Marseille	First Prolog Implementation
1977	D.H.D. Warren	Edinburgh	First compiler (DEC-10 Prolog)
1980?	K. Clark	London	MicroProlog developed
1981	K. Fuchi	Tokyo	Fifth-generation project announced
1982		Marseille	First International Conference on Logic Pgmg.
1982	F. Pereira	Edinburgh	C-Prolog distributed
1984		Atlantic City	First U.S. Symposium on Logic Programming
1985	J.A. Robinson	Syracuse	Journal of Logic Programming started
1984	D.H.D. Warren	Palo Alto	Quintus founded
1986		London	Association for Logic Programming started

What is the syntax of a Prolog program?

There are two parts:

A set of *clauses* (relatively static)

A *query* or *goal* (which starts things off)

The primitive *syntactic* construct of the above is the Prolog *term.*

A *term* is either:

A numeral, *e.g.* 5, -99, 6.23E-14

An *atom*, designated by a string of characters beginning with a lower case character, *e.g.* foo, b1234, x, aFAR, or enclosed in single quotes, *e.g.* 'Abraham Lincoln', 'X'.

A *variable*, designated by a string of characters beginning with an upper case character, e.g. X, Foo, A21, or beginning with the special character _.

A *composite term*, which has the form

constructor(*term*$_1$, *term*$_2$, ..., *term*$_n$)

where each *term*$_i$ is a term and *constructor* is an atom, *e.g.*

a(1)

baz(c, x23)

foo(bar(a), hiss)

are all composite terms.

Operator syntax:

Some functors can be alternatively expressed as *operators*, which in Prolog means that they may have infix, prefix, or postfix representations.

An example is the **+** functor: We can write a term as either

```
+(X, Y)
```

or as

```
X + Y
```

Some functors have a built-in operator syntax.

For others, operator syntax may be declared by the user.

Interpretation of Prolog terms:

Prolog terms are *abstract*; their meaning *depends on context*.

For example, they may mean:

Record construction and component selection.

```
employee(Name, Address, SSNo)
```

Expression to be evaluated (as in an arithmetic expression)

```
3*X + 5
```

(such an expression is actually evaluated only when the context so indicates)

Goals directing Prolog execution:

```
brother(john, X)
```

Prolog program fragments:

```
son(X, Y) :- parent(Y, X), male(X).
```

(In the above term, :- is the constructor with operator syntax.)

There are many options for

Representation of Knowledge in Prolog

Several types of representation are possible.

Example: Suppose we wish to represent the following bits of information :

A certain animal, x, is a rabbit.

x is white.

x has red eyes.

We could do this with various of the following terms:

rabbit(x)

x(rabbit)

is(rabbit, x)

is(x, rabbit)

white(x)

x(white)

has_color(x, whilte)

eye_color(x, red)

has_color(eyes(x), red)

red(x(eyes))

color(eyes, x, red)

etc.

The choice of representation will depend on the rules for using the knowledge.

Prolog clauses

The bulk of a Prolog program is usually a set of *clauses*. These are the counterpart of statements in most languages, but depending on their form, take meaning as:

Procedure definitions

Declarations of logical relationships

Data definitions

Facts

The simplest form of Prolog clause is called a *fact* (or *unit clause*). It has the form:

Prolog term.

The period following the term identifies this as a fact. Here is an example of several facts:

```
male(john).

parent(john, caroline).

parent(jackie, caroline).

female(jackie).

female(caroline).
```

Here we may interpret the atoms *john*, *jackie*, and *caroline* as individuals, and the facts assert *relationships* among the individuals.

Rules

The general form of Prolog clause is called a *rule*. A rule has the form

term **:-** *term*$_1$, *term*$_2$, ..., *term*$_n$.

The term on the left of **:-** is called the *head* of the rule. The other terms comprise the *body* of the rule.

The symbol **:-** can be read *if*, or *provided*. The meaning of the rule is that the term on the left hand side is *solvable* as a Prolog goal provided that the terms on the right hand side are all solvable, in the sequence indicated.

Example:

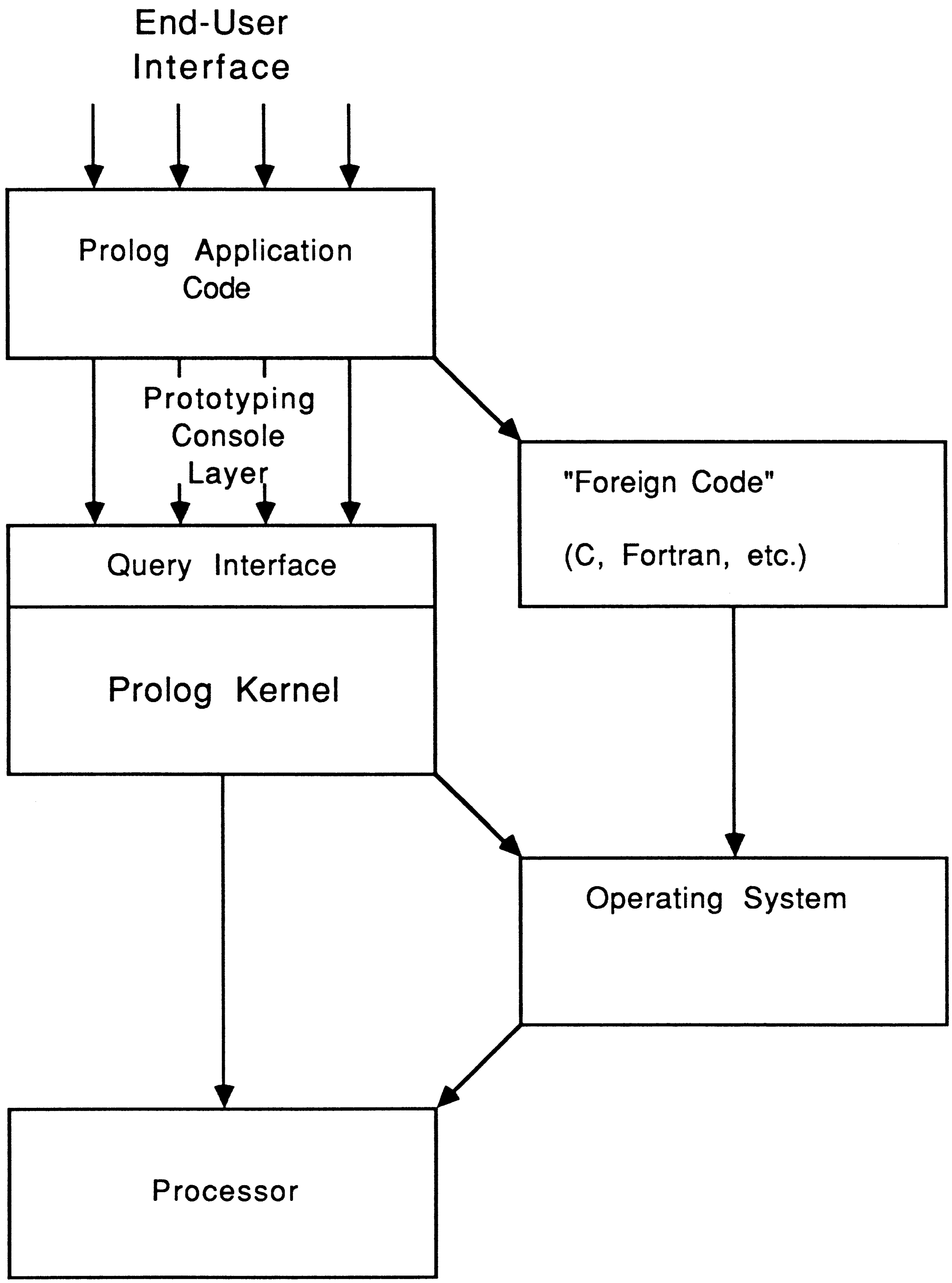
father(X, Y) :- parent(X, Y), male(X).

This asserts that a goal father(X, Y) is solvable if parent(X, Y) is solvable and male(X) is solvable.

We see that the commas separating the body terms can be interpreted as a logical *and*, while the symbol :- can be interpreted as logical *if* (in other words, the conjunction of the right-hand side terms *logically implies* the left-hand side).

Keep in mind that X and Y above are Prolog *Variables*, which means that they represent *arbitrary* entities. Furthermore, these variables are regarded as being *distinct* from variables of the same name in any other clauses.

Building Prolog Applications



Quintus Computer Systems, 1988

Prolog Database: The collection of all clauses for a program (facts and rules) is referred to as the *database.*

Example: Suppose that the database is:

```
male(john).
parent(john, caroline).
parent(jackie, caroline).
female(jackie).
female(caroline).

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

Ground terms: A term is called *ground* if it contains no variables. When a goal is a ground term, it is either solvable, or not, i.e. there is a **yes or no answer**. Prolog can determine solvability of the goal relative to the current database. This is done by entering the goal following the prompt ?-.

Consider Prolog's response to some typical ground goals:

```
?- female(jackie).
yes

?- parent(john, jackie).
no

?- mother(jackie, caroline).
yes
```

In the first two queries, Prolog simply only used facts in the database. In the third query, it had to use the rule for mother, which translates into the two goals

```
female(jackie), parent(jackie, caroline).
```

In this instance, the two goals are solvable by facts. In general, long chains of rules might be necessary to establish a goal.

Non-ground goals:

When a Prolog goal is non-ground, Prolog tries to find a way of solving it by substituting a value for the variables, if necessary. Such a substitution also called an *instantiation* or *binding*.

Examples: Assume again the database

```
male(john).
parent(john, caroline).
parent(jackie, caroline).
female(jackie).
female(caroline).

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

Here are Prolog's responses (in bold-face) to some non-ground goals:

```
?- male(X).
X = john

?- female(X).
X = jackie

?- parent(john, X).
X = caroline

?- parent(jackie, X).
X = caroline

?- parent(X, Y).
X = john,
Y = caroline

?- parent(X, X).
no

?- mother(X, Y).
X = jackie.
Y = caroline.
```

Multiple Solutions to a Goal:

In certain of the goals in the preceding example, we see that there are multiple ways in which the goal could be true in principle. For example,

```
?- female(X).
```

Here either X = jackie or X = caroline would satisfy the property.

When Prolog presents its answer to a query, if we follow by return, that is the only answer we shall see.

If we follow by semicolon, then we may see another answer (if there is one). We may repeat this response as often as desired.

```
?- female(X).

X = jackie   ;

X = caroline    ;

no
```

Builtin Predicates taking Predicate Arguments

Often we need to find *all solutions* to a goal query and collect them into a list. A special builtin predicate does this for us:

bagof(*term*, *goal*, *answer* **)**

computes all terms of the form *term* which satisfy *goal* and collects them into the list *answer*.

Example:

bagof(X, father(john, X), Y)

collects into list Y the set of john's children.

bagof([X, Y], father(X, Y), Z)

collects into list Z the set of list of pairs [X, Y] such that father(X, Y) is solvable

A similar predicate which *sorts* the result list according to the universal order is called *setof.*

Ergonomic uses of operators:

```
:- dynamic manages/2.
:- op(500, xfx, manages).
:- op(500, xfx, reports_to).
:- op(500, xfx, responsible_for).
:- op(400, fx, discharge).

john manages joe.
john manages jack.
john manages tim.
tim manages sally.
tim manages fred.
jack manages sue.
jack manages charles.

X reports_to Y :- Y manages X.

X responsible_for Y :- X manages Y.
X responsible_for Y :- X manages Z,
                       Z responsible_for Y.

discharge X :-
    retractall(X manages _),
    retractall(_ manages X).
```

Record construction and Extraction

The unification process is useful for building records and extracting their components. For example, we might have a rule:

```
overpaid(employee(Name, Salary, Level)) :-

        Salary > 10000 * Level.
```

When we try to solve a goal such as

```
        overpaid(employee(smith, 100000, 9))
```

the match which takes place instantiates the Name, Salary, and Level components of the record to smith, 100000, and 9 respectively.

A Prolog "procedure" is a collection of all clauses with a given head predicate.

The two interpretations of a Prolog procedure:

The *logical* interpretation: Each clause represents a logical implication. The collection of clauses, the heads of which match a given goal, define the only ways in which the goal may be true. For example, if the clauses for brother_in_law are:

```
brother_in_law(X, Y) :- brother(X, Z), spouse(Z, Y).

brother_in_law(X, Y) :- husband(X, Z), sister(Z, Y).

brother_in_law(X, Y) :- husband(X, Z), sister(Z, W), spouse(W, Y).
```

we mean

for arbitrary entities X, Y, Z, and W

```
        if (brother(X, Z) and spouse(Z, Y)) then brother_in_law(X, Y)
and
        if (husband(X, Z) and sister(Z, Y)) then brother_in_law(X, Y)
and
        if (husband(X, Z), sister(Z, W), and spouse(W, Y))
                then brother_in_law(X, Y)
```

The *procedural* interpretation:

To solve for a goal, try the first of the clauses having a head which matches the goal. Solve each of the corresponding body terms in turn. If this attempted solution fails, try the next relevant clause similarly, etc. If none of the clauses works, then the goal is unsolvable.

There are two ways in which information flows from a Prolog procedure:

Through the argument values

Through the *success* or *failure* of the procedure. Each time a procedure is called, it either succeeds or fails.

If it succeeds, then control proceeds as normal.

If it fails, then it causes Prolog to abandon the current calling sequence and seek an alternative within the next clause of the procedure. If there is no next clause, then the calling procedure also fails, etc.

A compiler for an Expression Language to a Stack Machine

Uses "Definite-Clause Grammar" notation

```
:- op(550, xfx, ':=').

comp((S;T)) -->
    {comp(S, CS, []), comp(T, CT, [])},
    CS, CT.

comp(A:=B) --> {atom(A)}, !,
    [addr(A)], comp(B), [store].

comp(A+B) --> !,
    {comp(A, CA, []), comp(B, CB, [])},
    CA, CB, add_inst.

comp(A*B) --> !,
    {comp(A, CA, []), comp(B, CB, [])},
    CA, CB, mult_inst.

comp(A) --> {atomic(A)}, !,
    [A], load_inst.

load_inst --> [load].
add_inst --> [add].
mult_inst --> [mult].
```

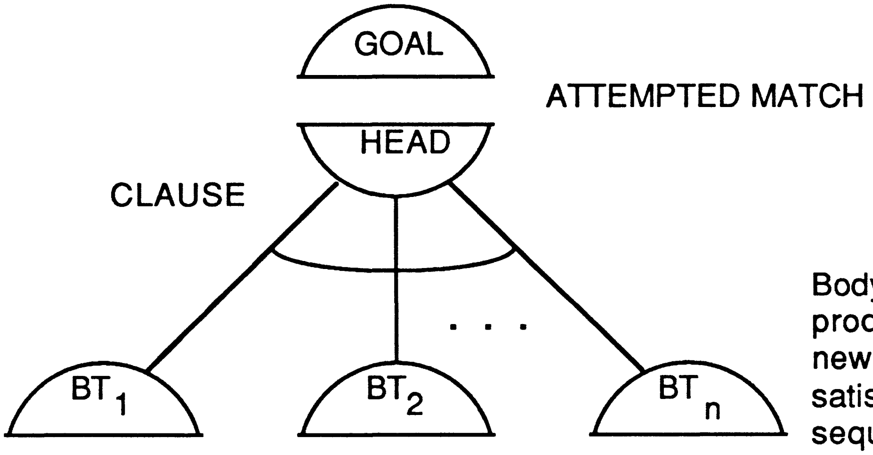
An Emulator for the Stack Machine

```
% calc(Instruction_List, Stack)

calc([], _).

calc([X,load | Y], Stack) :-
   integer(X), !,
   calc(Y, [X | Stack]).

calc([X,load | Y], Stack) :-
   calc(Y, [value_of(X) | Stack]).

calc([mult | X], [Y, Z | More]) :-
   compute(Y*Z, W),
   calc(X, [W | More]).

calc([add | X], [Y, Z | More]) :-
   compute(Y+Z, W),
   calc(X, [W | More]).

calc([addr(V) | X], Stack) :-
   calc(X, [addr(V) | Stack]).

calc([store | X], [Value, Addr | More]) :-
   write('store '), write(Value), write(' into '), write(Addr), nl,
   calc(X, More).
```

The AND-OR Tree View of Prolog Execution

Matching of a Goal with a Clause

head :- BT$_1$, BT$_2$, ..., BT$_n$.



Body terms
producing
new goals to be
satisfied
sequentially

In general, several clauses may match a given goal

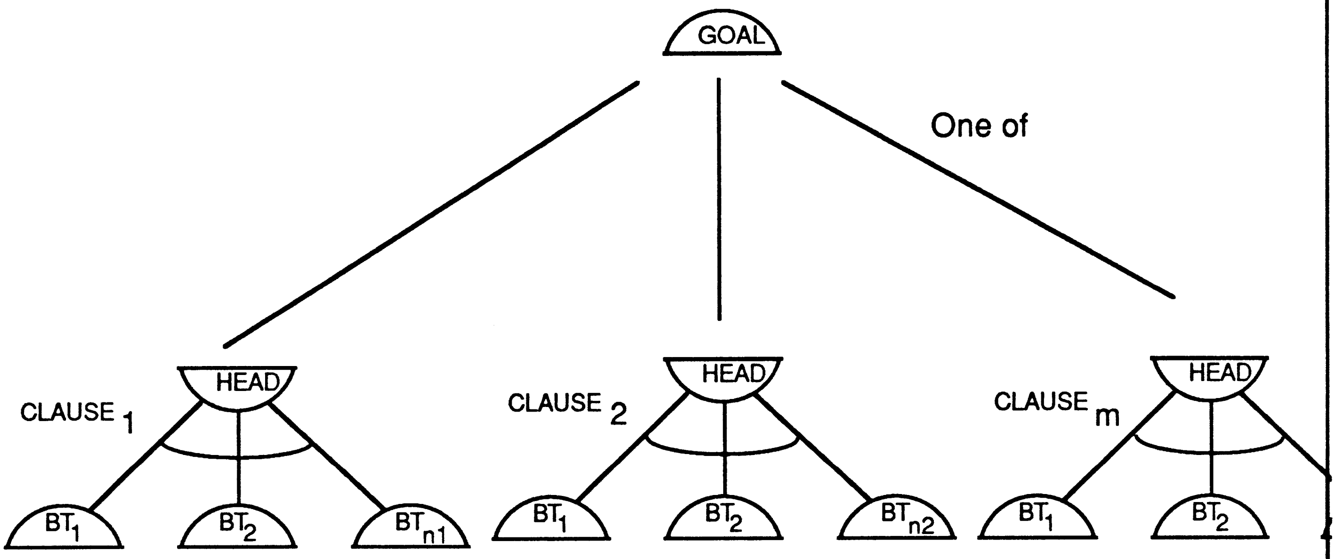
head :- BT$_{11}$, BT$_{12}$, ..., BT$_{1n}$.
head :- BT$_{21}$, BT$_{22}$, ..., BT$_{2n}$.

head :- BT$_{m1}$, BT$_{m2}$, ..., BT$_{mn}$.

In general, several clauses may match a given goal

head :- BT_{11}, BT_{12}, ..., BT_{1n}.
head :- BT_{21}, BT_{22}, ..., BT_{2n}.

head :- BT_{m1}, BT_{m2}, ..., BT_{mn}.



Backtracking

Backtracking occurs in any query where we ask to see multiple answers. In effect, we cause the answer just seen to act as if it were failure, and Prolog picks up its search where it left off.

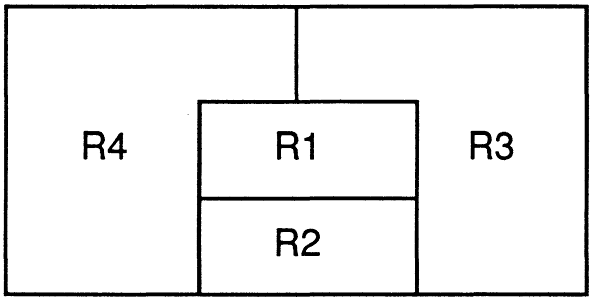
To further see the power of backtracking, consider the following **map problem:**

> *It is desired to color the regions from a set of colors on a map so that no two adjacent regions have the same color. Find a solution if one exists, or report that none exists.*

The simplest way to solve this problem is just to try various colorings until one is found to work, or until all have been tried. This is a complicated bookkeeping problem in a language without backtracking. In Prolog, it is simple and elegant.

Solving the map problem:

We express the map by using the regions as variables and a list of terms representing adjacencies. For example, the following map is expressed as shown:



```
map(R1, R2, R3, R4) :- adj(R1, R2), adj(R1, R3), adj(R1, R4),
                       adj(R2, R3), adj(R2, R4), adj(R3, R4).
```

We express the fact that each pair of adjacent regions must be colored, and must be colored by distinct colors, by:

```
    adj(R, S) :- color(R), color(S), distinct(R, S).
```

The predicate *distinct* is true exactly when its arguments are not the same. In Prolog, we could define it using the built-in predicate \== which tests that its arguments are not identical:

```
    distinct(R, S) :- R \== S.
```

We need to give a the list of colors, e.g.

```
    color(C) :- member(C, [red, blue, yellow]).
```

Then we need only present the goal

```
    ?- map(R1, R2, R3, R4).
```

and Prolog does the rest. How? Backtracking is the key.

Prolog-in-Prolog Meta Interpreter

```
solve(true) :- !.

solve((Goal, Goals)) :- !, solve(Goal), solve(Goals).

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

Remember here that *clause* will in general return a *Body* with some substitutions already made, as occur in the matching of *Goal*.

Caution: The above does not work if an interpreted clause contains *cut*. It is difficult to handle this case with the tools in standard Prolog.

Meta Interpreter which gives Reasons

```
solve(true, []) :- !.                                % no reason
needed

solve((Goal, Goals), [Reason | Reasons]) :-          % conjunction
        !,
        solve(Goal, Reason),
        solve(Goals, Resons).

solve(Goal, [(Goal :- Body) | Reasons]) :-           % rule
        clause(Goal, Body),
        solve(Body, Reasons).
```

General Problems in running "dusty" Prolog code in Parallel

As with many languages,

Prolog code is typically written with sequentiality in mind;

The Prolog programmer often optimizes his code by ordering subgoals. A prime example is in the

"generate and test"

problem solving method

```
generate(Input, Candidate),     test(Candidate).
```

Logically, the two subgoals can be reversed or done in parallel,
but in the Prolog execution model, there can be a variety of disasters:

Arithmetic variables must be bound in Candidate before testing.

test(Candidate) may be divergent if Candidate does not have a certain binding state, due to depth-first assumption.

test(Candidate) may have an infinite number of solutions for Candidate

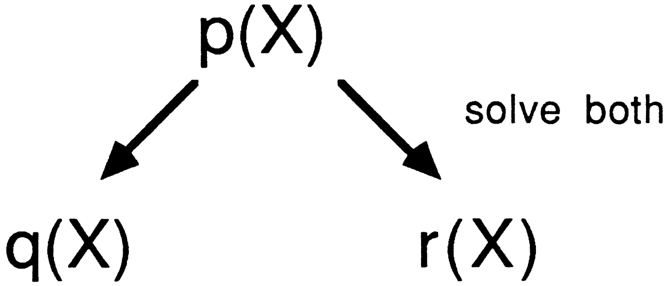
AND parallelism

Clause

p(X) :- q(X), r(X).

Modes (generally dynamic)

X ground



p(X)

solve both

q(X)

r(X)

Speculative, since if q(X) fails,

sequential execution would have been better

Wholesale AND Parallelism through Recursion

forall(Predicate, List) means the Predicate is true on each element of List

Clauses

```
forall(_, []).

forall(Predicate, [A | X]) :- call(Predicate, A),
                              forall(Predicate, X).
```

Usage

```
forall(p, [x1, x2, x3, ...., xn])
```

can spawn parallel executions of

```
p(x1), p(x2), p(x3), ...., p(xn).
```

Similar possibilties hold for predicates such as map:

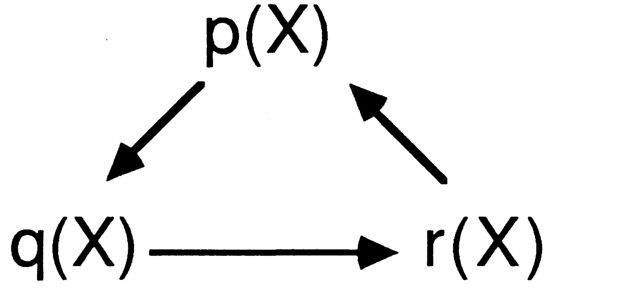
map(Predicate, List1, List2) means that Predicate is called pairwise on elements of List1 and List2 (usually the predicate is functional).

AND parallelism (cont'd)

Clause

p(X) :- q(X), r(X).

X not ground



p(X)

q(X)

r(X)

on success,
binding can "flow" forward

If multiple solutions for p(X) are desired,
this flow yields "pipelining".

In principle, q(X) and r(X) could go in parallel above,
but then there would be the problem of

reconciling bindings

AND parallelism (conclusion)

Knowledge (by the compiler or execution system) of two aspects of the data are essential:

Knowing whether terms are ground

Knowing whether there are any logical variables shared between terms.

Without these, the overhead to get correct execution may be too high.

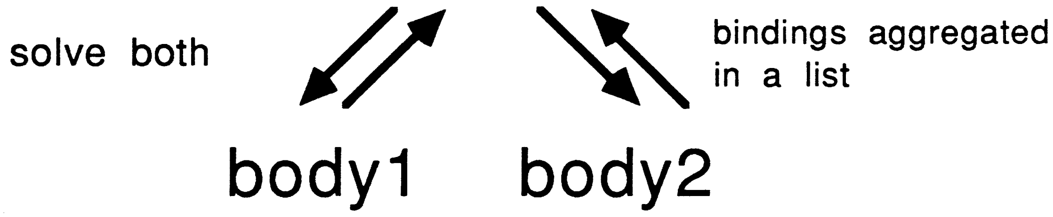
OR parallelism

Clauses

```
p(X) :- body1.

p(X) :- body2.
```

Cases of Interest:

Find all solutions to p(X):



solve both

bindings aggregated in a list

body1 body2

OR parallelism (cont'd)

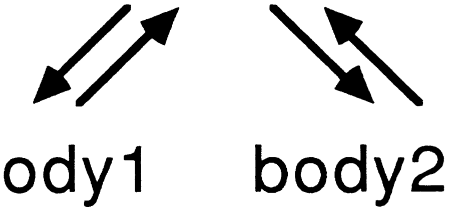
Clauses

```
p(X) :- body1.

p(X) :- body2.
```

Cases of Interest:

Find the first solution to p(X):

solve both



body1 body2

As above, but must synchronize bindings.

Each body may itself have multiple solutions, so the overall parallelism is an unlimited "tree" of execution.

Additional parallelism possible in Anticipation of need for additional solutions.

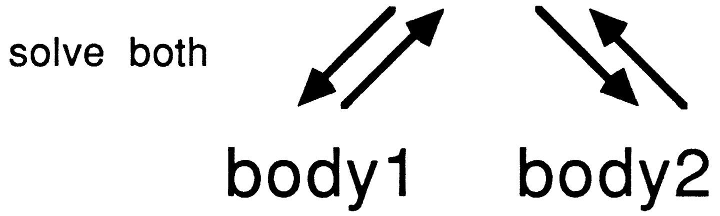
OR parallelism (cont'd)

Clauses

```
p(X) :- body1.

p(X) :- body2.
```

Cases of Interest:

Find a solution to p(X):



Truly non-deterministic result.

Speculative

Super-linear speedup possibilities

Proper Prolog semantics will have uses for the first two modes, but not the third, which would require a special directive to avoid incorrectness.

Implementation Problems for OR parallelism:

Maintaining large numbers of separate variables (for binding in parallel) while sharing bindings wherever possible (to economize space).

Synchronization in parallel OR branches, when side-effects are involved

Aurora

Aurora - a prototype Prolog system exploiting OR-parallelism

"Workers" explore the Prolog search tree in OR-parallel

- the "engine"
- the "scheduler"

The Aurora implementation environment:

- Engine-scheduler interface
- Scheduler test harness
- Instrumentation

Conclusions from Aurora

Engine overhead due to SRI model and scheduler hooks: 15-35%

This overhead defines breakeven with sequential systems

Speedups Measured under Aurora:

	speedup for N workers		
Example	3	5	
parse5	(2.83)	(4.08)	!!
8-queens2	(2.97)	(4.88)	!!
salt&must	(2.87)	(4.82)	!!
parse3*20	(2.09)	(2.30)	??
farmer*100	(1.63)	(1.69)	??

Speedups measured on a six processor Sequent Balance

Selected References on Parallelism in Prolog

A. Ciepielewski and S. Haridi. *A formal model for the OR-parallel execution of logic programs*,. IFIP '83, North-Holland.
keywords: OR-parallelism

J. S. Conery. *Parallel execution of logic programs*, Kluwer Academic Publishers (1987).
keywords: AND-parallelism, OR-parallelism

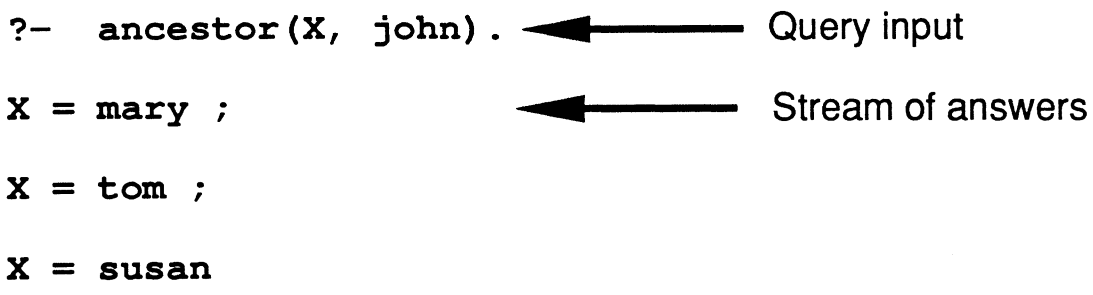
T. Disz, E. Lusk, and R. Overbeek. *Experiments with OR-parallel logic programs*. Proc. Fourth International Conference on Logic Programming, 576-600, J.-L. Lassez (ed)., M.I.T. Press (1987).
keywords: OR-parallelism

M.V. Hermenegildo. *An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel.* PhD Thesis, University of Texas at Austin (August 1986).
keywords: AND-parallelism

Y-J. Lin. *A parallel implementation of logic programs.* PhD Thesis, University of Texas at Austin (May 1988).
keywords: AND-parallelism

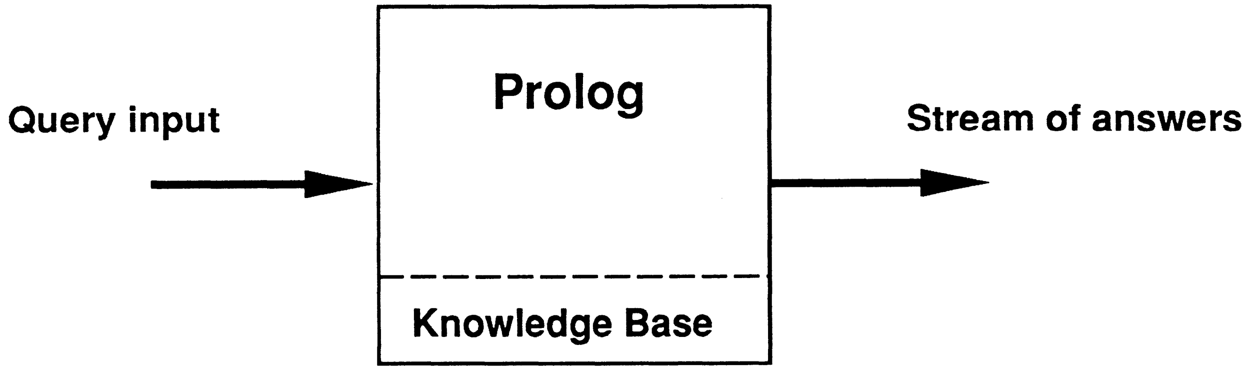
D.H.D. Warren. *OR-Parallel execution Models of Prolog.* Tapsoft '87, Volume 2, 243-259, LNCS 250,. Springer-Verlag.
keywords: OR-parallelism

Knowledge-Server Viewpoint

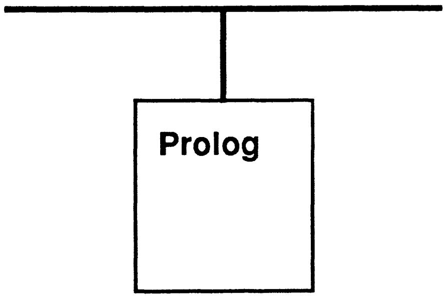


```
?-  ancestor(X, john).

X = mary ;

X = tom ;

X = susan
```

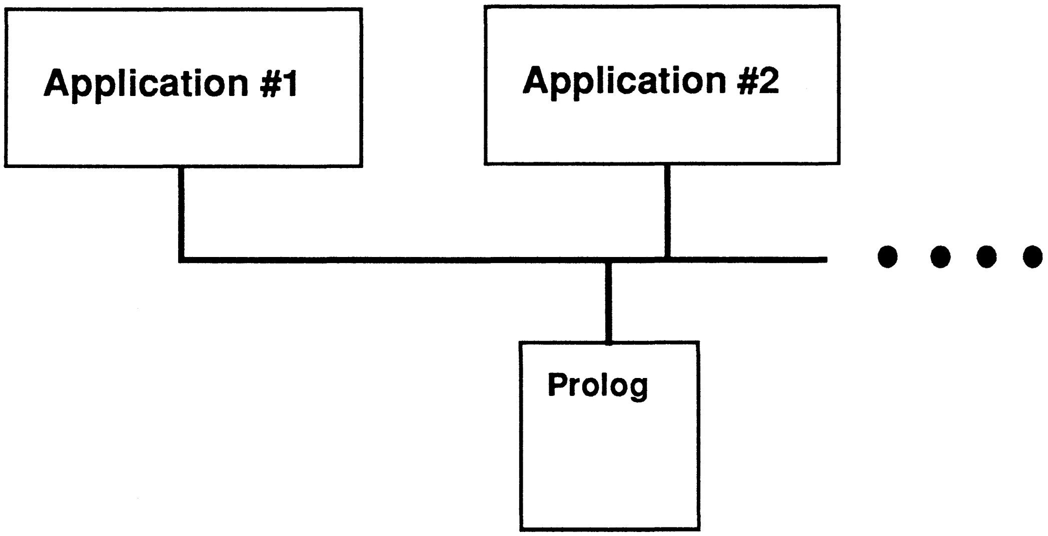
Schematic View

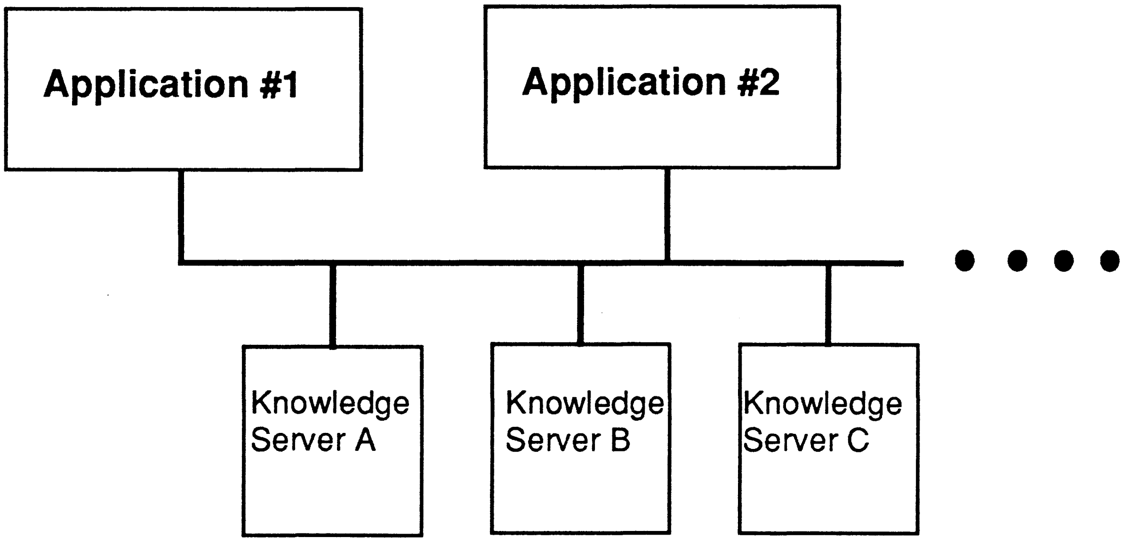


Simplified "Bus-Oriented "Representation



Exploiting the "Knowledge Net"


Application #1
Application #2
Prolog


Application #1
Application #2
Knowledge Server A
Knowledge Server B
Knowledge Server C