

PROLOG Introduction

Prolog is a programming language in which one lists a number of statements asserted to be true. These form the Programs "Data Base". Included in the Data Base are statements of facts and statements of legitimate inference. The statements of fact are given in the form of relations declared to be true, and inferences as the declaration that if a sequence of relations are true, then a given relation must be true. All these assertions involve variables and constants, so "larger(5,2)" and *brother(sam, george)* can represent relations between constants: of two actual numbers, and two actual people respectively. The names 'brothers' and "larger" are implicitly defined by their use in the Data Base (so that the relation *larger(5,2)* means simply that if queried *?larger(5,2)* Prolog would say true, whereas the query *?-larger(7,6)* would not be true without more in the Data Base, (If in addition to the *brother(sam, george)* *sibling(george, mary)*" were in the data base these by themselves would not imply the truth of *sibling(sam, mary)* despite the conventional meaning of these words. However if the implications:
sibling(A,B) :- brother(A,B), meaning if A and B are brothers they are also siblings and
sibling(A,C) :- brother(A,B), sibling(B,C) meaning if (A and B are brothers) and (B and C are sibling) then (implies) (A and C) are siblings, then *sibling(sam, mary)* would be true that is the Query *?- sibling(sam, mary)* would result in a YES response. Furthermore the Query *?- sibling(sam, X)* would result in *X = george* and also *X = mary*. In reading these statements the capitalized words are read as variables and lower case as constants. So Prolog not only will answer YES or NO given a query involving constants, but will assign variables in the Query values which would make the Query true.

As an aid to understanding, Data Base assertions can often be described in a functional manner. This may be helpful particularly for recursive implications. So the definitions of sibling and brother above can be functionally paraphrased"

sibling(A,C) :- brother(A,B), sibling(B,C) is a such a recursive statement. These are broadly equivalent to functional statements. For this case the functional form

Prolog

brother(sam, george)

/ is equivalent to brother (sam) = george.*/*

sibling(george, mary)

/ is equivalent to sibling((george) = mary.*/*

sibling(A,C) :- brother(A,B), sibling(B,C) */* is equivalent to sibling(A) = sibling(brother(A)).*/*

In the examples of Data Bases assertions, the "functional paraphrase", though not part of Prolog, is often included-as a comment which may clarify the given assertions.

Prolog is a much more powerful than Imperative and Functional languages. It can generate results without explicitly providing a procedure for obtaining that result. Instead one need only declare the relevant data that implies the result. So, as we show later, one can give the rules in a regular grammar in Prolog form in the Data Base. These are almost identical copies to the usual way of writing such rules, and this would be sufficient to parse (give the sequence of rule applications) a given string. The Parser, a finite state machine, in this case, need not be designed. Also given a Data Base with an assertion (which can be paraphrased with a recursive functional form) in Prolog, one can often get the inverse. So, for the example above, one could query *sibling(a, X)* and also *sibling(X,b)* and get the value of X in both cases. Prolog achieves its power by its ability to try one path toward a proof and if that fails, try another-it operates in "try and fail" mode. On the other hand this also makes it a very inefficient means of getting the desired answers. Its good for explorations but not for production.

In general, a perfect Logic language, will be able to answer any Query if it can be proven given the assertions in the Data Base. Prolog doesn't always do that because its proof technique is not completely general. To be completely general would make the, already relatively slow implementation prohibitively long. On the otherhand, Prolog has commands which can shorten the general search for an answer in the name of efficiency. The *cut!* operation can be used to limit (trying alternative paths) backtracking..

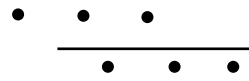
The Data Base of Prolog consists of the conjunction (ands) of Horn Clauses. The Horn Clauses are implications. Each Horn Clause is equivalent to disjunctions (ors) of Atomic formulas. Atomic Formulas are the relations we have given examples of above (ex. *brother(A,B)*)

A :- B, with B = B₁, B₂, . . . , B_n. is a Horn Clause meaning if “B₁ & B₂ & . . . & B_n” are true then A is true. or equivalently (the truth of) B₁ & B₂ & . . . & B_n. **implies** the (truth) of A where B_i is assertions of the truth of relations, ex. (A > 5) brother(ec, sam). To show the generality of the Horn Clause we derive an equivalent form for a Horn Clause in terms of “&’ (.) and “ors’ (+)

A	B	A :- B
0	0	1
0	1	0
1	0	1
1	1	1

$$A :- B = A + \overline{B} \text{ (if } B (\overline{B}=0) \text{ true then } A \text{ must also be true (} A=1))$$

$$= A + B_1 \cdot B_2 \cdot \dots \cdot B_n \text{ if } B = B_1 \cdot B_2 \cdot \dots \cdot B_n$$



Truth Table Definition Of Implication = A + $\overline{B_1} + \overline{B_2} + \dots + \overline{B_n}$ is also a Horn Clause

The Data Base is the Conjunction of Horn Clauses, i.e., of Disjunctions and this is a Canonical or Universal Form, namely, one in which in which virtually any logical statement can be expressed.

Example

weather(monday,fair).
weather(tuesday,overcast).
weather(wednesday,fair).
weather(thursday,fair).
weather(friday,rain).
weather(saturday,rain).
weather(sunday,fair).
previous(monday,tuesday).
previous(wednesday,thursday).
previous(sunday,monday).
color(sky,blue,Day) :-
weather(Day,fair).
happy(birders,Day) :-
weather(Day,fair),
active(birds,Day).
happy(birders,Day) :-
observed(rarebird,Day).
active(birds,Today) :-
previous(Today,Day),
weather(Day,fair).
observed(rarebird,tuesday).
observed(rarebird,thursday).

“and”- Implications Form

A₁.
A₂.
.
.

A_m.
A₁₁ :- B₁₁, B₁₂, . . . , B_{1n}.
A₁₂ :- B₂₁, B₂₂, . . . , B_{2n}.
.
.

A_{1p} :- B_{p1}, B_{p2}, . . . , B_{pn}.

A_{1m} :- B_{m1}, B_{m2}, . . . , B_{mn}.

Disjunctive Normal Form

= *A₁.*
 = *A₂.*
.
.

 = *A_{1m}*
 = *(A₁₁+ $\overline{B_{11}}$ + $\overline{B_{12}}$ + . . . + $\overline{B_{1n}}$).*
 = *(A₂₁+ $\overline{B_{21}}$ + $\overline{B_{22}}$ + . . . + $\overline{B_{2n}}$).*

 = *(A_{p1}+ $\overline{B_{p1}}$ + $\overline{B_{p2}}$ + . . . + $\overline{B_{pn}}$).*

 = *(A_{m1}+ $\overline{B_{m1}}$ + $\overline{B_{m2}}$ + . . . + $\overline{B_{mn}}$).*

| ? - *happy(birders,When).* General | ? Q₁, . . . , Q_n Queries

When = tuesday, thursday, [observed]
saturday,wednesday [weather fair and active-previous] _

The Prolog Data Base

As stated above the Horn Clause 'A :- B₁, B₂, . . . , B_n.' **Means:** if B₁ & B₂ & . . . & B_n. are all true **then** A is true. Notice that the reordering the terms B₁, B₂, . . . , B_n does not change the logical meaning of the Horn Clause since & (.) is commutative ex., John is smart & John is tall is **true iff** John is tall & John is smart. is true. Also the Horn Clauses in the data base are connected by &s so their reordering does not change the logical statements made by the Data Base ex. if *weather(tuesday,overcast).* and *weather(wednesday,fair)* in the example above were interchanged it would not change the meaning of that Data Base . This is the case for a pure Logic Language. Prolog is not quite that. Some rearrangements may cause problems

Early Warning

In the Data Base form given previously A_{ij} and B_{ij} are Atomic Formulas and have the form ' $rel(X_1, \dots, X_m)$ ' where ' rel ' names a relation (also called a predicate) ' rel ' is Builtin or User Defined. Builtins ' rel 's are often represented by conventional symbols (ex. $>$, $<=$) and refer to the relation usually associated with those symbols. There are also many other Builtins.

BUILTINS:

As we said the arithmetic relations $>$, $<$, $<=$ are built in.

In addition the operators $+$, $*$, etc. are builtin. There are also Builtin constructs for operations on Lists. We will assume that $head(L, X)$ meaning the first member of list L is X , and $tail(L, T)$ meaning T is the list consisting of the second through last member of L . These are easily defined based on Builtins. How this is done together with a more extensive list of Builtins is given on pp 12-13.

For User Defined relations the name given to the relation has no implicit meaning-instead it is defined by the Data Base occurrence of that name-these may relate constants or more generally describe the conditions which establish that relation.

Atomic Formulas		Examples	
binary relation	relation(X_1 , X_2)	$A >= B$ - variable A is greater or = to variable B $(10 > A)$ - number 10 is greater than variable A	Builtin
		$parent(mary, bill)$ - $mary$ has the relation ' $parent$ ' to $bill$	User Def
trinary relation	relation(X_1 , X_2 , X_3)	$med(3, 7, 10)$ - The med relation relates 3, 7 and 10 $med(A, M, D) :- A <= M, M <= D . - \text{if } \underline{A} <= \underline{M} \ \& \ M <= D$ the med relation holds between $A, M,$ and D -but I have in mind $- \text{if } \underline{A} <= M <= D$ them M is the median of A and D .	

The formal meaning of a Horn Clause is simply a statement about names of relations, those of which are builtins have their usual connotations, while those not builtin remain merely names all of whose meanings are determined by Horn Clauses in which they appear in the Data Base. It is important to remember this. However when one writes a Clause one has in mind some connotations of the named relations and the meaning of the Clause can probably be more understandably expressed using those connotations. So for the following definitions we have included what we have in mind

$cons(X, L1, L2) :- head(L2, X), tail(L2, L1) - \text{if relation } head \text{ holds for } L2 \text{ and } X, \text{ and } tail \text{ for } L2 \text{ and and } L1$
then relation $cons$ holds amongst $X, L1$ and, $L2$.

Meaning: if the head of $L2$ is X & the tail of $L2$ is $L1$ then cons of X with $L1$ yields $L2$

where we assume the head of a list = its first member and the tail of a list = the remainder of the list after the first member is remove, are builtin predicates.

Following are a set of Clauses intended to give an operative definition of the highest number in a list of numbers

$max(M, X, M) :- M >= X,$

Means: Maximum of M and X is M if M is greater or equal to X /* $max(M, X) = M$ if $M >= X$ */

$max(M, X, X) :- X > M,$

Meaning: Maximum of M and X is X if X is greater than M /* $max(M, X)$ if $X > M$ */

$highest([], M)$

Meaning: The highest number found in an empty list and M is M /* $highest(L, M) = M$ if $L == []$ */

$highest(L, M, Y) :- head^1(L, X), tail^2(L, T), max^3(X, M, Z), highest^4(T, Z, Y) \text{ /*highest(L, M) =}$

Meaning: The highest number in list L and the number M is Y^0 $highest(tail(L), max(head(L), M) \text{ */}$

if X is the head of list L^1 and T is the tail of L^2 and Z is the maximum of M and X^3 ,

and the highest of the numbers in T and the number Z is Y^4 and $M >= Q^5$,

English Interpretations Of Atomic Formula and Horn Clauses In Prolog

As shown another Way To Express the Meaning of a Prolog Data Base, particularly for recursive Horn clauses, is the Functional Paraphrase. So summarizing the functional paraphrase above.

$max(M, X) = M$ if $M \geq X$, $max(M, X) = X$ if $X > M$, $highest(L, M) = M$ if $L == []$ $highest(L, M) = highest(tail(L), max(head(L), M))$	Functional Paraphrase
--	------------------------------

This functional paraphrase is a function definition which gives the highest number of those numbers in a list L and the number M. It is deeply nested since each function call returns a value. The equivalent Prolog statement of these functions do not return values to the call but rather represent Procedures in which a value computed by the Procedure is assigned to a parameter. For example

highest(L, M, Y) does the same computation as *highest(L, M)* .but returns its value to Y instead of returning it to the call.

tail(L, X) returns the value of *tail(L)* to X, etc.

Doing this un-nests the definition to give the Prolog form

The Query *highest([1, 5, 7, 3, 2], 0)* returns the correct value just as
 The Query is *? highest([1, 5, 7, 3, 2], 0, Y)* , will give the correct value of Y.
 In A Pure LOGIC LANGUAGE the Query:
 | ? *highest([1, 5, 7, 3, 2], M, 20)* should result in $M=20$, and
 | ? *highest([1, 5, 7, 3, 2], M, 7)* should give $M=0 ; M= 1; \dots; M=7$
 That would correspond in the functional form query :
 | ? *highest([1, 5, 7, 3, 2], M) = 7* giving
 $M=0 ; M= 1; \dots; M=7$

Queries

In general a Query has the form of a conjunction of Atomic Formulas-most simply a single Atomic Formula. The arguments of the Queries may consist of constants only or may contain variables. For Queries containing only constants the response is either True if the resulting conjunction is true by derivation from the Data Base or False otherwise. For queries with constants the value of the constants for which the conjunction is true are given if there are such, otherwise it is False. This is the case for an ideal Logic Language-because of Prologs implementation it may not always be the case for more complex Horn Clauses This is why it is necessary to have some understanding of the Prolog implementation.

<p>DB <i>loves(al, marilyn).</i> <i>loves(al, al).</i></p> <p>Queries ? <i>loves(al, al)</i> T ? <i>loves(al, harry)</i> F ? <i>loves(al, X)</i> X=<i>marilyn</i> X=<i>al</i> ? <i>loves(Z, al)</i> Z = <i>al</i> ? <i>loves(Z, marilyn)</i> Z = <i>al</i></p>	<p>DB <i>t(b, c).</i> <i>t(a, b).</i> <i>t(A, C) :- t(A, B), t(B, C).</i></p> <p>Queries ? <i>t(X, c)</i> X=<i>b</i> ? <i>t(b, c), t(Z, c)</i> Z=<i>a</i> ? <i>t(b, c), t(Z, d)</i> F</p>
--	---

In general meanings of statements in the Data Base have been interpreted in terms of as an ideal Logic Language. Largely for practicality (reasonable running times) Prolog is only an approximation to such a language. The meaning of the data base should remain unchanged if the Horn clauses are reordered, or the Atomic formula on the right of a Horn Clause are reordered. Since Prolog orders its test for proof from the first to last Horn Clause and from left to right in the Atomic Formulas on the right of Horn Clauses results do depend to some extent upon ordering. Two ways of defining concatenation are given below, only differing in the ordering of their Atomic Formulas, but resulting in different outcomes when a proof is attempted to respond to a query.

The predicates $head(L1, X)$, $tail(L1, T1)$, $cons(X, Q, L3)$ have been defined on the previous page. $concat([], L2, L2)$.

$concat(L1, L2, L3) :- head(L1, X), tail(L1, T1), cons(X, Q, L3), concat(T1, L2, Q)$.

Means: if the head of L1 is X & the tail of L1 is T1 & cons of X and Q is L3 & concat of T1 L2 is Q then concat L1 with L2 yields L3

$concat1([], L2, L2)$.

$concat1(L1, L2, L3) :- head(L1, X), tail(L1, T1), concat1(T1, L2, Q), cons(X, Q, L3)$

Means: if the head of L1 is X & the tail of L1 is T1 & concat1 of T1 L2 is Q & cons of X and Q is L3 then concat L1 with L2 yields L3

In pure Logic Language These would be equivalent. But in Prolog they are not

$concat$ IS GOOD with virtually any query, BUT $concat1$ LOOPS WITH A VARIABLE AS THE FIRST ARGUMENT, ex. $concat1(X, [x], [y])$, SHOULD SAY NO, BUT IT LOOPS. (The recursive call is last in $concat$).

concat1: if L1 is a variable then head leaves X a variable, tail leaves T1 a variable. So $concat1$ only knows L2. The recursive call of $concat1$ then again head and tail give no additional information and when $concat1$ is called again it is the same shape, only knowing L2.

concat: if L1 is a variable then head leaves X a variable, tail leaves T1 a variable. So $concat$ only knows L2. Next construct, knowing L3 can generate X, and so the recursive call of $concat$ knows L2 and Q and can compute T1. $Construct$ is good for inverses since knowing any of its arguments all others are determined.

$concat(L1, L2) = L2$

if $L1 = []$

$concat(L1, L2) = cons(head(L1), concat(tail(L1), L2))$ if $L1 \neq []$

Functional Paraphrase

Note: The Functional Paraphrase of $concat$ and $concat1$ are the same.

The Prolog Interpreter, "Inference Engine"

The set of Horn Clauses in a logic program, called the **Data Base (DB)**, give all the *facts* and *implications* and thus all the logical postulates by which query responses are justified. The query (Q) itself is part of the **DB** in this declarative language with its powerful built in **Inference Engine (IE)**, serves the same related purposes as the **Program (P) + input (I)** and its **Compiler (C)** do in a procedural language [**DB + Q \leftrightarrow P + I, IE \leftrightarrow C**]

As noted Prolog is not a perfect logic program. It has both Declarative (purely logical) and Procedural aspects. For Prolog to answer queries correctly it is necessary that its Data Base give the information to provide that answer logically - but that is not enough. The implementation, depending as it does on the order of the DB, will miss some logical conclusions since logical derivation should not depend on that order. **so the implementation can miss logically sound inferences**

Though Prolog does not employ a perfect Inference Engine, it provides a tool for solving many problems with much shallower knowledge than would be necessary to solve the same problem with a Functional or imperative language. This is at least partially due to the fact that it can handle non-determinism in a complete specification of a problem because it typically test many paths to a solution - having failed one it can back up and try another. In a functional or imperative language one needs a deterministic solution. An example is provided by a regular grammar which is to be parsed by a finite state machine. Suppose the RG is

1 Algorithm For Answering Queries - The Inference Engine

The following algorithm works for clauses in which the atomic formulas have arguments which are either constants or variables. In Prolog, other complex arguments such as those for lists, are allowed. Our algorithm will however be mainly confined to cases in which the arguments are simple constants and variables. The algorithm is a model of the Prolog implementation (inference engine). Though it may not accurately describe any such implementation, it is adequate to describe the trace provided by Prolog for cases involving the arguments covered, as well as implementation dependent functions such as the *cut (!)*.

Outline: At each stage in evaluating a query, a logic program examines a Conjunction of Atomic Formulas called the **CAF**. Initially the CAF = the input query. One of the atomic formulas, referred to as the Call, is selected for evaluation. In Prolog the leftmost call is always chosen.

Search, Match First there is a search through DB for a Clause, Cls, whose head, ClsHd, is Unifiable with Call.

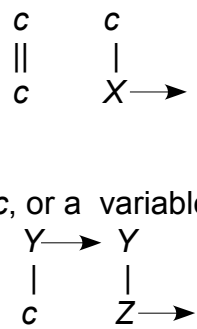
Given our restriction to variable and constant arguments, these clauses are potentially unifiable.

- (a) Call and ClsHd have the same predicate,
- (b) the same number of arguments, and
- (c) any argument constant in both is the same in both. (This leaves Variables in one or both)

Rename Before unification, all variables in Cls are effectively renamed to insure that Cls and CAF have no variable names in common.

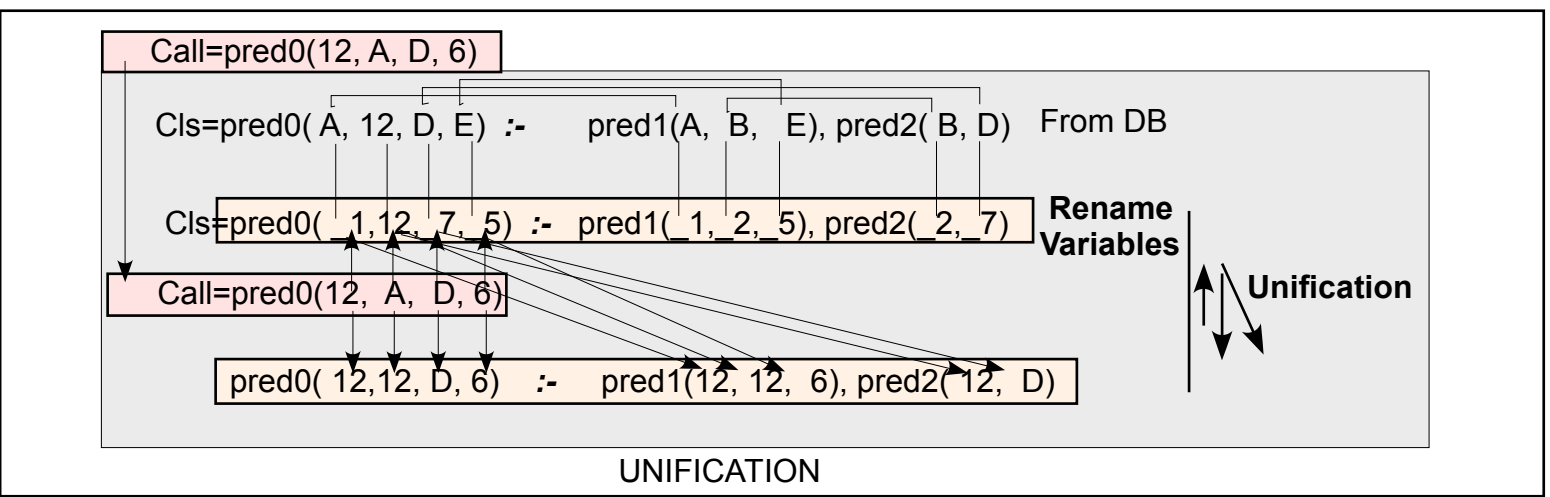
Unification:

- (a) If the *i*th argument of Call is a constant, *c*, ClsHd's then the *i*th argument is either
 - i. the same constant *c* or a variable.
 - ii. If it is a variable, *X*, then *X* is effectively replaced by *c* throughout the Cls, which certainly remains a true clause.
- (b) If the *i*th argument of Call is a variable, *Y*, then the *i*th of ClsHd is either a constant, *c*, or a variable, *Z*.
 - i. In the first case *Y* must effectively be given the value *c* throughout CAF,
 - ii. In the second *Y* effectively replaces *Z* throughout the Cls.



We have just outlined a unification procedure which by itself is adequate for most simple cases. Before giving a more detailed procedure which more closely reflects the actual traces given in Quintus Prolog we briefly justify the procedure.

[Justification Of Unification In Inference Engine Model The clauses in the data base are all assumed to be true. If a variable in one of these clause, *s*, is replaced with a constant throughout the result is still true. Also if a variable is renamed throughout the result is still true. If an atomic formula *A* (the Call) is to be proven true from the truths in DB this can be done if either there is an atomic formula *A* in DB, or a clause *A :- B* and *B* can be proven true. But there may also be an atomic formula *A'*, or an implication *A' :- B* in which, though *A'* is not identical to *A*, it can be unified to be identical to *A*. That is, by substituting constants or renaming variables in *A'* (and consistently throughout *B*) the result will *A* be and still true. Alternatively it may be possible to change variable in Call *A* (and consistently throughout the CAF) to constants or to similarly rename variables so that the resultant Call is identical to *A'*. This has no effect on our goal unless constant substitutions are involved in which it does change the goal, but only by replacing it with more specific case of the same goal.]

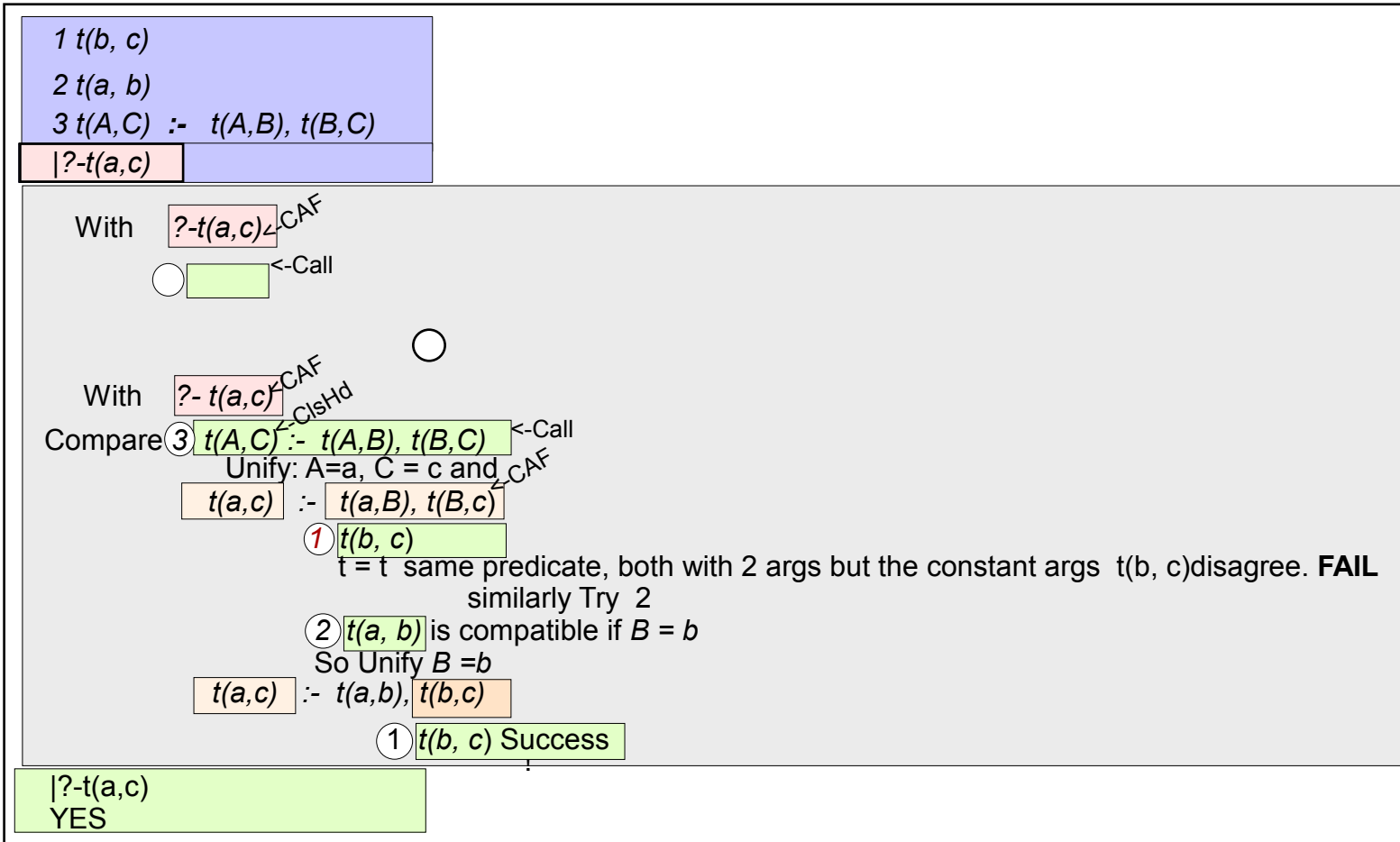


Formation of the next CAF to be proven.

After successful Unification substitution have been made in the original CAF and in the Clause whose head was matched with the call. The next CAF to be Matched and unified is the unified right side of the Clause and the RU(Clause) and the unified part of the CAF following the Call, RE(CAF). The new CAF to be Matched and unified is RU || RE. Note that if Clause consists only of a Cause Head, the resultant CAF will be shorter than the original CAF.

If the procedure continues with CAF-Match-Unify-New CAF until the CAF disappears. The proof is successful and the result is YES if all arguments of the original Query (CAF) are constants. If some of these arguments are variable, these would generally have been given constant values in the unifications in the proof. The result in these cases is the list of these variables and their constant value

Here is a simple example of a successful derivation



Backup

If at any point the Formulation of the next CAF fails it is necessary to Backup to the Call for the failing unification Call0 and consider unification with the clause head following the one that failed to unify with the Call. If all such attempts follow it is necessary to backup further to the Call-1 the one prior prior to Call0 and consider clauses beyond the one that last unified with Call-1. If this procedure goes back to the first Call and fails. The resultant Query results in the **result No**.

Tree Style Trace Derivation

With this initial development of the explanation of the trace we are in a position to show more complete traces. These are given in a form different than that given by the Prolog interpreter, but somewhat simpler.

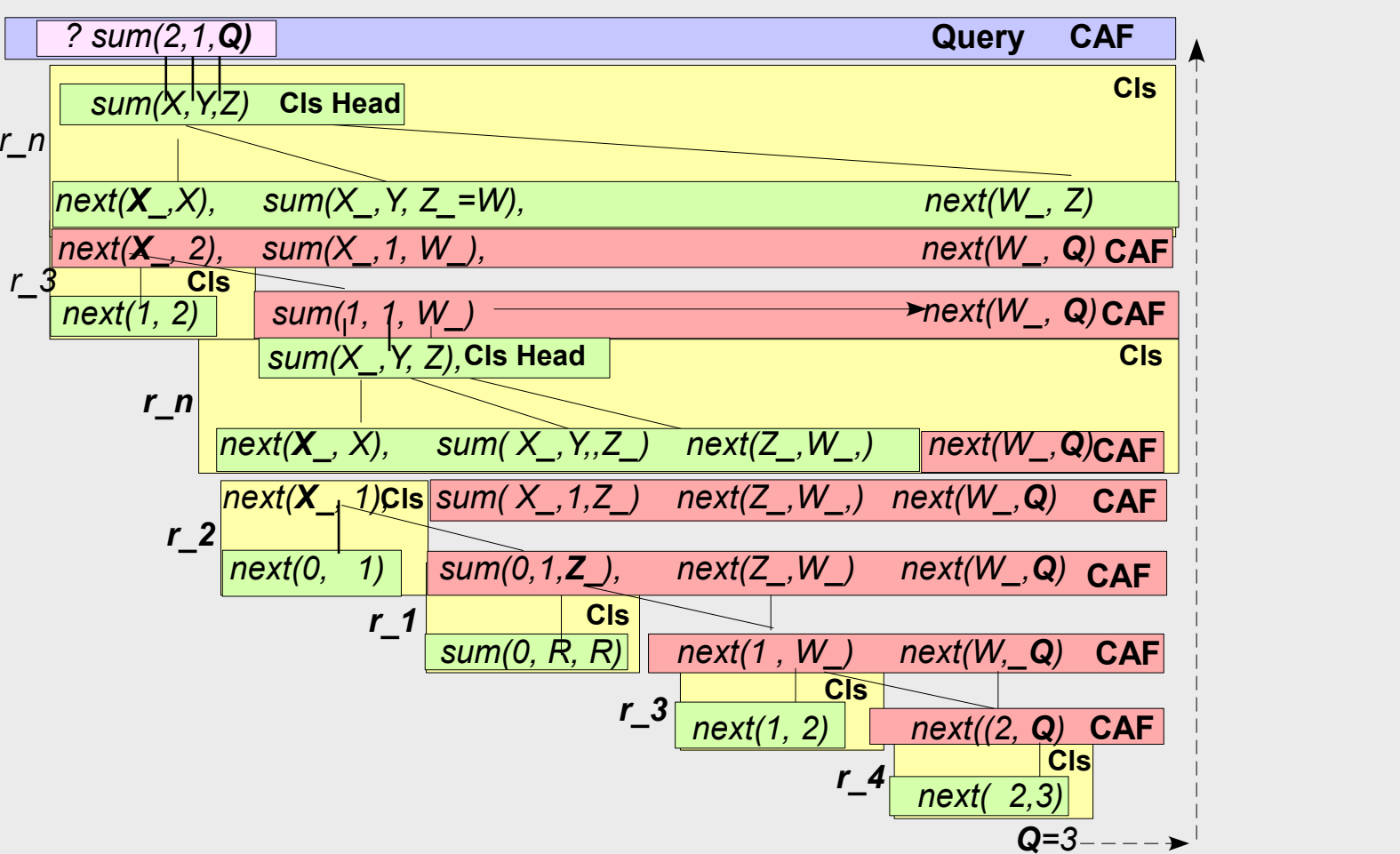
The derivation by which Prolog answers a query is analogous to a derivation in a CFG. Right sides of rules are repeatedly substituted for left sides. Substitution is always into the leftmost unsatisfied atomic formulas. Here there is also unification which can alter the right side substituted or the entire expression into which it is substituted. Also here there can be back-up. Despite these differences, the ways to represent a derivation which is successful, or one which ends with a backup correspond to those for a CFG. So to represent the complete set of actions of the response of Prolog to a query is by an ordered set of such derivation trees.

We can represent the derivation by the series of CAF's that result from the series of substitutions or by a tree in which the sequence of terminal nodes at each step represent the current CAF.

In the following set of figures in we have illustrated the tree representation of a derivation for a number of simple data bases and queries.

The basic subtrees represent **Clauses** or rules which match the goal atomic formula of the **CAF's**. They contain a parent,, which is the **Head of the clause** and its children,, which are the right side of the clause. The Clauses Head, its children and the resultant CAFs are color coded. In the first example derivation we show a series of successful substitution of unified right sides of clauses. The first CAF is the root of the tree Then the clause whose head is at the root of the tree is unified and its unified children become the next CAF. The leftmost of this CAF is the goal and it s substituted for with the clause with head The third CAF is also shown with the dotted line at 3. The current CAF in the tree is always the current sequence of terminal, nodes of the tree. The tree as shown continues to completion with clause or rule i. If there were f ailure of all possible substitutions at a node backup is to the previous node at which substitution was succesful. So failure at node f would cause backup to node e, failure at e cause backup to d, etc.

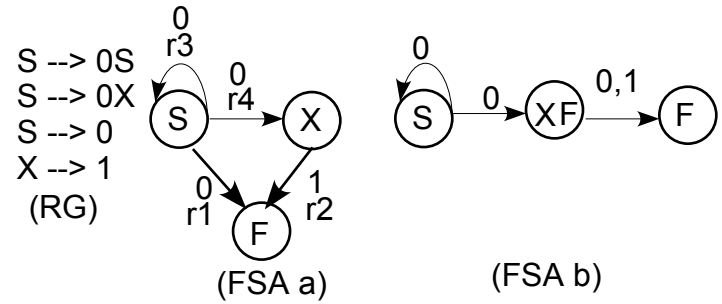
$r_1 \text{ sum}(0,R,R)$ $r_2 \text{ next}(0,1)$ $r_3 \text{ next}(1,2).$ $r_4 \text{ next}(2,3)$ $r_n \text{ sum}(X, Y, Z) :- \text{next}(X_, X), \text{sum}(X_, Y, Z_), \text{next}(Z_, Z)$	Data Base	$\text{sum}(0,R)=R$ $r_1 \text{ next}(0) = 1 \quad r_1' \text{ next}^{-1}(1) = 0$ $r_2 \text{ next}(1) = 2 \quad r_2' \text{ next}^{-1}(2) = 1.$ \vdots $\text{sum}(X,Y) = \text{next}(\text{sum}(\text{next}^{-1}(X), Y),$	Functional Paraphrase
---	------------------	---	------------------------------



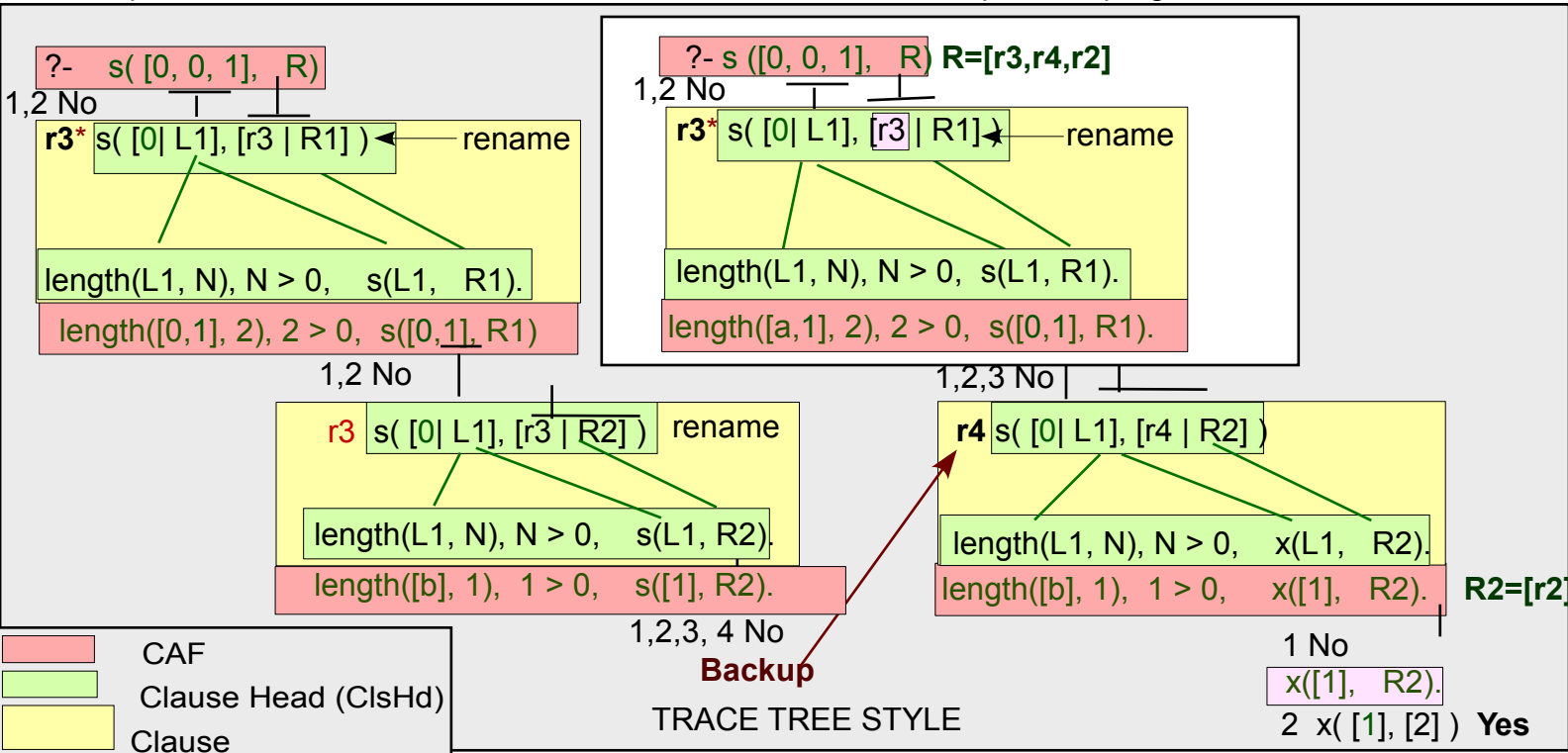
Definition Of Addition With Simple Data Base
TRACE: TREE STYLE

A parse must give a sequence of Rules (ri s). This is a FSM non-deterministic, although we can get a deterministic machine which accepts the same language with a standard transformation (a) that will not give a correct parse. To get a deterministic parser it is necessary to look-ahead in the input string in this simple case just 1 input ahead. From (a) anytime we get a 0 the rule r1, r3, and r4 are all candidates-if the next two input are 0s then r3 should be used if the next two are 01 then X should be the next state, and if there is only one more input, namely a 0 then r1 is the next rule used. Despite all this a simple DB which consists basically of the RG rules is sufficient to respond to a Query about any input string. There are RGs whose parses require considerably looking ahead further even than the number of NTSS in that RG. In general for many problems which have a non-deterministic characteristic susceptible to a "Trial and Failure" approach the formulation of the necessary DB is simpler to specify than a deterministic Functional program, the formulation of the latter requiring a deeper understanding of the problem than its statement. On the other hand the Functional program, avoiding failure and backup will generally run faster.

1 s([0], [r1]).	(RG DB)
2 x([1], [r2]).	
3 s([0 L1], [r3 R]) :- length(L1, N), N > 0, s(L1, R).	
4 s([0 L1], [r4 R]) :- length(L1, N), N > 0, x(L1, R).	
? s([0, 0, 1], R)	
f(IN, k, X, 0, #, R) = R r1	Functional Paraphrase
f(IN, k, S, 0, 1, R) = R r4 r2	
f(IN, k, S, 0, 0, R) = f(IN, k+1, S, 0, IN(k+2), R r3)	



So a Functional Parser, which is not simply a simulation of the logic language's "Trial and Fail" approach requires the above analysis and the the Functional program which would incorporate "lookahead". This functional program would be faster in producing parses than the Prolog implementation. This occurs for many other problems so a Prolog DB can be formulated and experiments can be performed on in relatively rapidly, but for a production solution a Functional or even more efficient, an Imperative program would be the best



The next example uses a simplified form of the Tree Trace to show how reordering the same set of Clauses in the DB can effect the result of Prologs Inference Engine, even causing it to fail (infinite loop).

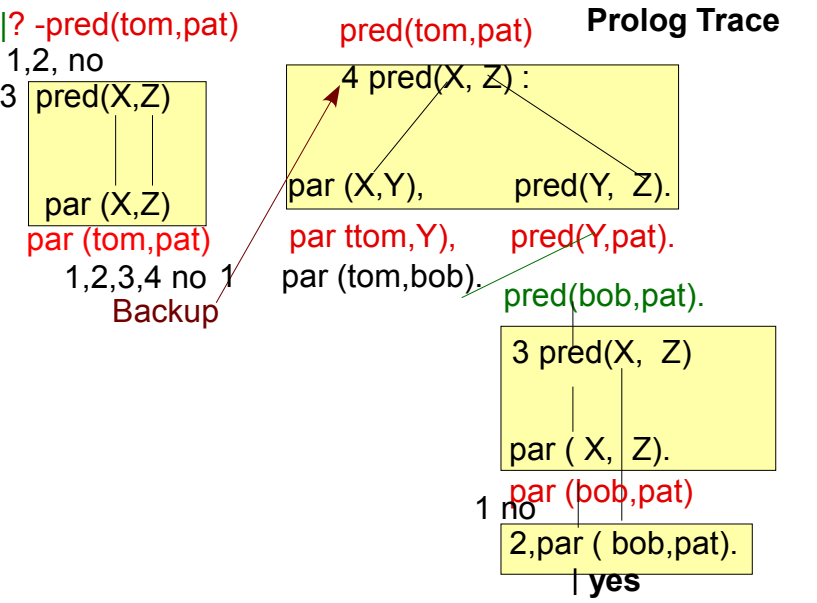
Prolog I

```

1 par(tom,bob).
2 par( bob,pat).
3 pred(X,Z) :- par(X,Z).
4 pred(X,Z) :- par(X,Y), pred(Y,Z).
    
```

Functional Paraphrase	&Trace
1 par(tom) = bob.	? pred(tom)= pat ,
2 par(bob) = pat.	3 par(tom)= pat , no
3 pred(X) = par(X).	4 pred(par(tom)).= pat ,
4 pred(X) = pred(par(X)).	3 pred(par(tom)). = pat
	1 pred(bob). = pat 3
	par(bob).= pat 2 yes

Functional Trace General
 pred(X) = pred(par(X))..... 4
 pred(X) = par(par(X))..... 3 in 4
 pred(X) = pred(par(par(X)).. 4 in 4
 pred(X) = par(par(par(X)).... 3 in 4 in 4



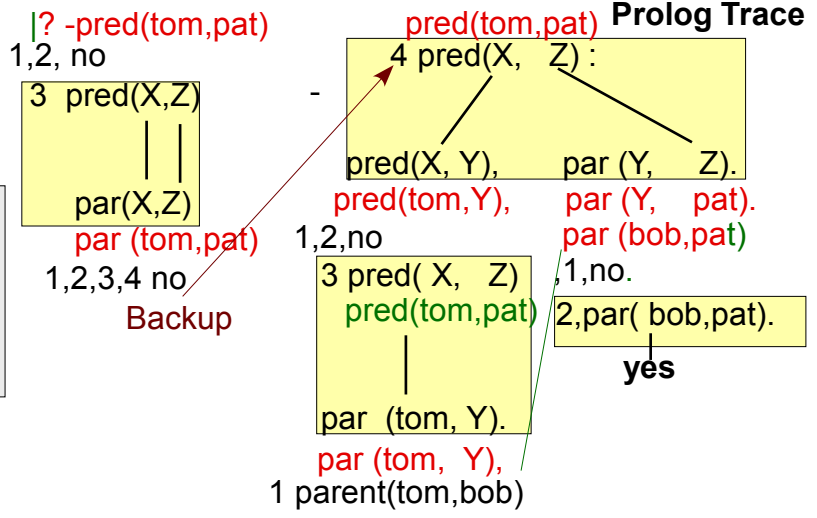
Prolog II

```

1 par(tom,bob).
2 par( bob,pat).
3 pred(X,Z) :- par(X,Z)
4 pred(X,Z) :- pred(X,Y), par(Y,Z). (Switched)
    
```

Functional Paraphrase	&Trace
1 par(tom) = bob.	?pred(tom)= pat ,
2 par(bob)= pat.	3 par(tom)= pat no.
3 pred(X) = par(X).	4 par(pred(tom)).= pat
4 pred(X) = par(pred(X)).	3 par(par(tom)). = pat
	1 par(bob).= pat 2 yes

Functional Trace General
 pred(X) = par(pred(X))..... 4
 pred(X) = par(par(X))..... 3 in 4
 pred(X) = par(par(pred(X)).....4 in 4
 pred(X) = par(par(par(X))..... 3 in 4 in 4 etc



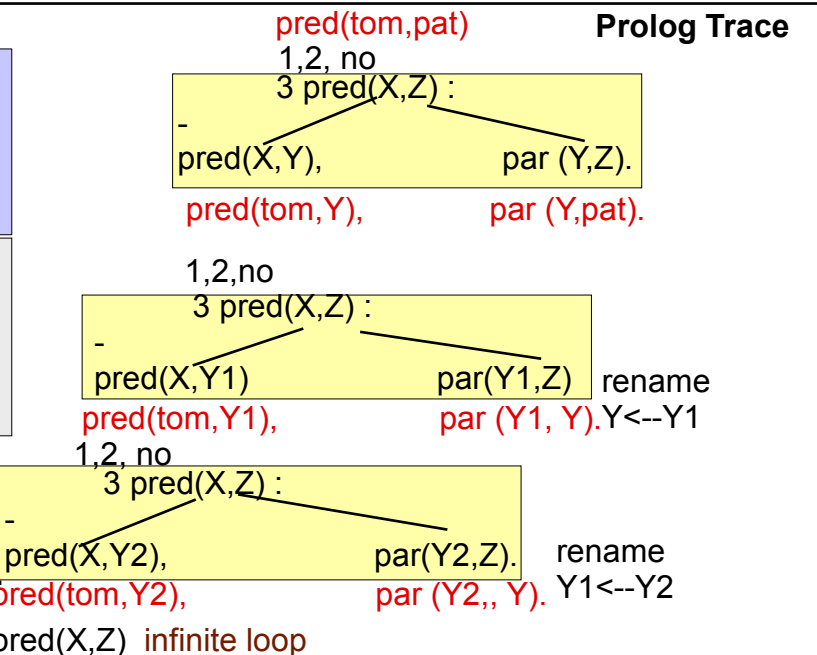
Prolog III

```

1 par(tom,bob).
2 par( bob,pat).
3 pred(X,Z) :- pred(X,Y), par(Y,Z).
4 pred(X,Z) :- par(X,Z) (Interchange)
    
```

Functional Paraphrase
1par(tom) = bob.
2 par(bob = pat.
3 pred(X) = par(pred(X)).
4' pred(X) = par(X)

Functional Trace General
 pred(X) :- par(pred(X)). 3
 pred(X,Z) :- par(par(pred(X))). 3 in 3
 pred(X,Z) :- par(par(par(pred(X))). 3 in 3 in 3 etc.



Clause CAF

THE EFFECT OF CLAUSE ORDER ON PROOF IN PROLOG

1 Example Trace

The following example consist of an actual traces which I have augmented with explanatory lines and comments. All lines with parenthesized numbers up to the first question mark are from the actual trace. The parenthesizing mentioned in the last part of the algorithm description is omitted, but thelevel is given by the numbers appearing to the right of the parenthesized numbers on the lines from the actual trace.

This DB Describes An FSM For Parity.

If it is started in state e, any input with an even number of 1's will bring it back to e, while a string with an odd number of ones take it to state o.

DATA BASE

$t(e,1,o)$

$t(e,0,e)$

$t(o,1,e)$

$t(o,0,o)$

$from-to(e, [], e).$

$from-to(o, [], o).$

```

CAF:   from-to(o,[0, 1], _60) ?
Cls:   from-to(A,[B|C], D) :- t(A,B, E), from-to( E, C, D ).
Cls:   unify from-to(o,[0|1]], _60) :- t(o,0, E), from-to( E, [1], _60).
Cls:   from-to(o,[0|1]], _60) :- t(o,0,_182), from-to( _182,[1], _60).
CAF:   t(o,0, _182) ? from-to( _182,[1], _60)
Cls:   (4) 1 Call: t(o,0, o) ?
Cls:   (4) 1 Exit:t(o,0,o) ?
CAF:   (5) 1 Call:
           from-to(o, [1 |[]], _60) ?
Cls:   from-to(A,[B| C], D) :- t(A,B, E), from-to(E, C, D ).
Cls:   unify from-to(A, [1 |[]], _60) :- t(o,1, _243), from-to(_243,[], _60)
CAF:   t(o,1, _243) from to(_243,[], _60)
           t(o,1,e) ?
Cls:   (6) 2 Call
Cls:   (6) 2 Exit: t(o,1,e) ?
CAF:   (7) 2 Call: from-to(e, [], _60) ?
Cls:   (7) 2 Exit: from-to(e, [], e ) ?
           grp( _60,e)
           Success. Return.
           Replace _60 with e.
           Success. Return.
           Replace _60 with e
Cls:   (5)1 Exit: from-to(o, [1], e ) ?
           (3) 0 Exit: from-to(o,[0,1],e) ?
R = e      Since at Success R was assigned _60.

```

TRACE PROLOG STYLE

QUERY

$from-to(o, [0,1],R).$

$from-to(A,[B|C],D) \Leftarrow t(A,B,E),from-to(E,C,D).$

$from-to(A,[B|C]) = from-to(t(A,B,E), C).$

$t(e,1,o)$ = if M is in state e(even) and receives a 0 it goes to state o(odd) . $from-to(e, [], e).$ = if M is in state e(even) and there is an empty input sequence [](none) M will go to state e

$from-to(A,[BjC],D) \Leftarrow t(A,B,E),from-to(E,C,D)$ if single input B takes M from state A to state E and input string C takes M frim state E toi state D, then input string [B|C] takes M form A to D.

0.1 Built In Functions

So far Prolog looks like a language with no built in functions-it looks as though everything can be defined given the basic schemes for defining facts and implications, using constants read literally, together with variables. Together with the recursive schema these are very powerful, but for practicality some builtins are needed.

Data Structure Terms:

Constants include

1. **uninterpreted atoms**: start with lower case or is an integer
2. **negative and positive integers, and floating point numbers** (with or without exponent notation (E))

3. **List Constants & Their Names:**

```
rarebird([condor,kiwi,dodo]          ). [condor,kiwi,dodo] is the list of rarebirds
?- rarebird(dodo)    no
?- rarebird(Dodo)   Dodo = [condor,kiwi,dodo]
```

Variables:

1. **Simple Variables** begin with capital letter and
2. **Structured List Variable:** $[H | T]$ for a variable which is a list, with H its first member and T the list without its first member.

```
?- rarebird( [ H | T ] ). H = condor, T = [kiwi,dodo]
```

Note that this structured variable we can build 2 predicates which can replace this notation:

```
head([H|T],H). H is the head of list L = [H|T]
```

```
tail([H|T],T). T is the tail of list L
```

```
member( [ H | T ], H ) . H is a member of a list whose 1st member is H
member( [ H | T ], X, ) :- member( [ T ], X ) if X is a member of [T] then X is a member
of a list with 1st member H followed by list T
```

```
[ ?- member(X, [x,[ [y,e],z] ]) X = x; X=[y,e], X=z
```

With the two list relations given above, the definition of membership could have been

```
member(L,H) :- head(L,H)
```

```
member(H,L) :- tail(L,T), member(T,H).
```

The basic unit of Prolog is the atomic formula. It is a predicate and it is either true or false. There are a large number of built in predicates: Some examples: A and B can be variables, constants, or functions or compound terms in variables and constants but, at the time the predicate is evaluated they must all be evaluated. A and B must be a single variable. Evaluate A and B (must have arithmetic result).

```
A ::= B T if val(A) = val(B), else F.
```

```
A /= B T if val(A) != val(B), else F.
```

```
A < B T if val(A) < val(B), else F.
```

```
A > B T if val(A) > val(B), else F.
```

```
A =< B T if val(A) =< val(B), else F.
```

```
A >= B T if val(A) >= val(B), else F.
```

The next example uses a simplified form of the Tree Trace to show how reordering the same set of Clauses in the DB can effect the result of Prolog's Inference Engine, even causing it to fail (infinite loop).

If V is a variable or a constant, E an arithmetic expression (must have an arithmetic result)

$V \text{ is } E$ If V is a constant then V is equal to $val(E)$ then T , else F . If V is a variable then V is unified with the value of E , and the predicate is true.

Other functions:

integer(X) T if X is currently instantiated to an integer, else F .

float(X) T if X is currently instantiated to a floating point number else F .

var(X) T if X is an un-instantiated variable, else F .

In addition to constants atoms and numbers and lists, variables, there are functions that can serve as terms in a predicate or relation. $+$, $-$, $*$, $/$, mod . These can be written as infix operations.

Finally there are many library functions, ex., *subtract/3 disjoint/2, subset/2*.

Special Relations

Cut = ! A cut can appear on the right of the a clause as follows:

$\alpha \text{ :- } \beta, !, \gamma$

. When α unifies with the goal in the CAF (say when the CAF is in state S) $\beta, !, \gamma$ (unified) replaces the goal in the CAF. Then when the cut is first reached it always succeeds. However if there is a backup to the cut, the cut fails {but more: Backup goes immediately to state S , and all attempts to match α also fail at this point and backup goes to the make the occurrence of α in the CAF in state S fail

1 Example Rules

SOME DEFINITIONS-SOME PROBLEMS-Functional Paraphrases

As we have seen like head and tail can be defined without using structures like $[]$, $[H | T]$, $[H | _]$ Here are alternate definitions of head and tail and definitions of *empty* list, and an equivalent to Scheme cons

empty($[]$). *is the LIST WE'VE NAMED empty*
head($[H | _]$, H). *H is the head of the list of the form H followed by _ (any list)*
tail($[_ | T]$, T). *T is the tail of a list whose firstmember is _ (anything)*

construct AND *construct1* (like Scheme *cons*) DO THE SAME THING-

construct(X , $L1$, $L2$) :- *head*($L2$, X), *tail*($L2$, $L1$). *if X is the head of List L2 & L1 is the tail of list L2 then [X] concatenated with L2 construct (form) L2 is true if ap-3 = a list with head= X and tail= L1*
construct1(X , $L1$, $[X | L1]$).

in AND *in1* DO THE SAME THING-OK WITH VARIABLES ANYWHERE, AND AFTER an

Note *in* and *in1* are the similar to the *member* predicate defined earlier,-the arguments are reversed.

in(X , L) :- *head*(L , X); *if X is the head of L, it is in L or*
in(X , L) :- *tail*(L , T), *in*(X , T). *If T is the tail of L & X is in T then X is in L*

in1(H , $[H | T]$) *The first argument is the head of the list in the second argument or*
in1(X , $[H | T]$) :- *in1*(X , T) *if X is in T, the tail of the second argument, then X is in the list.*

$head([H _]) = H$	$in(X, L) = true$ if $(X == head(L))$ or
$tail([_ T], T) = T$	$in(X, tail(L))$
	$in(X, []) = false$

Functional Paraphrase