The Prolog Interpreter Algorithm

Dennis L. Feucht
Innovatia Laboratories

Prolog is an unusual language in that it has only one procedure, the Prolog interpreter. This interpreter carries out an inference procedure that results in a depth-first search through a database of facts and rules. A Prolog program is a set of these facts and rules, or *clauses*. By applying the facts to the rules, new facts are generated as conclusions of the rules. In Prolog, an example of a fact is:

mammal(goat).

or "a goat is a mammal." An example of a rule is:

mammal(X) :- has-hair(X), gives-milk(X).

or "*X* is a mammal if *X* has hair and gives milk." In the rule, *mammal(X)* is the head of the rule, or the conclusion, :- is read "if," and *has-hair(X)* and *gives-milk(X)* are goals, separated by commas. These goals are the rule *body*. For the head to be asserted as a fact, all the goals must first be proved to be facts. The comma indicates a conjunction among goals in the body. A fact is merely a rule without goals. That is, no further goals must be proven true for the head of a fact to be asserted.

To illustrate the interpreter, **SEARCH**, an example list of clauses that is the program is:

1 *a* :- *x*, *y*.
2 *x*.
3 *y* :- *u*,*v*.
4 *u*.
5 *a* :- *b*, *c*, *d*.
6 *b* :- *g*, *h*.

To implement **SEARCH**, three lists are needed:

CLAUSES – contains the program, the list of rules and facts expressed in list form. The clauses above would be, in parenthesis notation for lists,

((*a x y*) (*x*) (*y u v*) (*u*) (*a b c d*) (*b g h*))

The head of the clause is the first item (or, also, *head*) of the list. The rest of the list (the *tail*) is the body of the clause.

GOALS – contains the goals to be proven. Initially, **GOALS** contains the desired goal(s) to be proven. Here, it will be (*a*).

SOLVED – contains a trace of the inferences performed to prove the goal and provides backtracking information in case search for a goal fails. It holds a copy of **GOALS** and a pointer into **CLAUSES**.

With these three lists, **SEARCH** works as follows. (See the algorithm for **SEARCH**.) A pointer to the clauses list is passed to **(SEARCH)** and it will return a flag. If it is true, the search was successful and the goal proven.

(SEARCH) is initially passed pointers to **GOALS** and **CLAUSES**. First, **GOALS** is checked. If it contains no goals (is empty or *nil*), then it returns a succeed flag. Either all the given goals were proven, or else there were none. (It is trivial to prove nothing from something). **FIND-CLAUSE** is then called. It scans down the clauses in search of a head that matches, or *unifies*, with the first goal in **GOALS**. If it unifies, the clause is *reduced*: a pointer, **^C**, into **CLAUSES** at the unified clause and a copy of **GOALS** are put on **SOLVED**. The body of the clause (if any) is appended to the head of **GOALS** after removing the first goal — the one that was just unified. Then **^C** is reset to (the head of) **CLAUSES**.

For example, if **GOALS** is (*a*), then the first clause in **CLAUSES** (given above) will match with *a*. The **SOLVED** list becomes:

SOLVED ← ((*a*) **^C**(*a x y*))

where **^C** indicates a pointer into **CLAUSES** to the rest of the **CLAUSES** list with (a x y) as the first item. The goals become:

GOALS ← (*x y*)

Since a matching clause was found for *a*, it is removed from **GOALS**. After this, **SEARCH** iterates, and **FIND-CLAUSE** seeks a match of *x* with a clause. Eventually, facts must be present to terminate the search since they add no further goals. If search fails, then other clauses further down in **CLAUSES** must be sought in an attempt to find an alternative sequence of inferences that succeeds. Continuing the example, the sequence of inferences are indicated by the states of **GOALS** and **SOLVED**:

	GOALS	**SOLVED**
1	(*a*)	()
2	(*x y*)	((*a*) **^C**(*a x y*))
3	(*y*)	((*x y*) **^C**(*x*) (*a*) **^C**(*a x y*))
4	(*u v*)	((*y*) **^C**(*y u v*) (*x y*) **^C**(*x*) (*a*) **^C**(*a x y*))
5	(*v*)	((*u v*) **^C**(*u*) (*y*) **^C**(*y u v*) (*x y*) **^C**(*x*) (*a*) **^C**(*a x y*))

At this point, no clause matches with *v* and this search sequence (or *path*) fails. **SEARCH** now *backtracks* to the previous state — that of matching with *u* — and seeks a different clause. To backtrack, the goals of state 4 are appended to GOALS and the pointer into **CLAUSES** is reset to clause 5. Also, the first two items on **SOLVED** that record this previous state are removed. Then search for an alternate clause matching *u* proceeds. (This search for *u* once again might seem redundant since a matching *u* was already found. When matching involves logic variables, different variable substitutions might occur with a different *u*, leading to a successful unification with a previously failed *y*.)

Since there are no more clauses that will match with *u*, backtracking once again occurs. This sequence of backtrackings continues until state 1 is reached. Then, an alternative match for goal *a* is found: (*a b c d*). Once again, reduction is invoked, but eventually this path also fails.

This pattern of activity can be graphically illustrated in the form of a *deduction tree*, as shown. The **REDUCE** path adds goals to **GOALS** and generates the **AND** levels of the tree. The **BACKTRACK** path generates the **OR** levels. Level 1 is an **AND** level because the goals all must be proven. (In this case there is only one goal, *a*, so it is a trivial conjunction of goals.) At level 2, two alternative clauses for *a* are shown. **SEARCH** will always try the leftmost branch in the tree first (since the clauses to the left appear first in **CLAUSES**). When (*a x y*) fails, (*a b c d*) is tried. Either (*a x y*) or (logical or) (*a b c d*), if successful, would cause **SEARCH** to succeed. The arc drawn between branches denotes an **AND** level.

The inference path of **SEARCH** for the given clauses is marked on the tree in circled numbers. **REDUCE** causes traversal down the tree while **BACKTRACK** causes upward traversal, in search of an alternative path that **REDUCE** can traverse downward. This traversing moves down and up from left to right.

For a complete Prolog interpreter, **UNIFY** must be the unification algorithm that provides substitutions for variables. In the explanation given here, only atoms (no variables) were used, making **UNIFY** trivial. Also, variable substitutions must be kept track of (on **SOLVED**); three items per state are required.

This algorithm uses iteration for both **AND** and **OR** levels. Recursion can be used instead for the **AND** level or both levels.

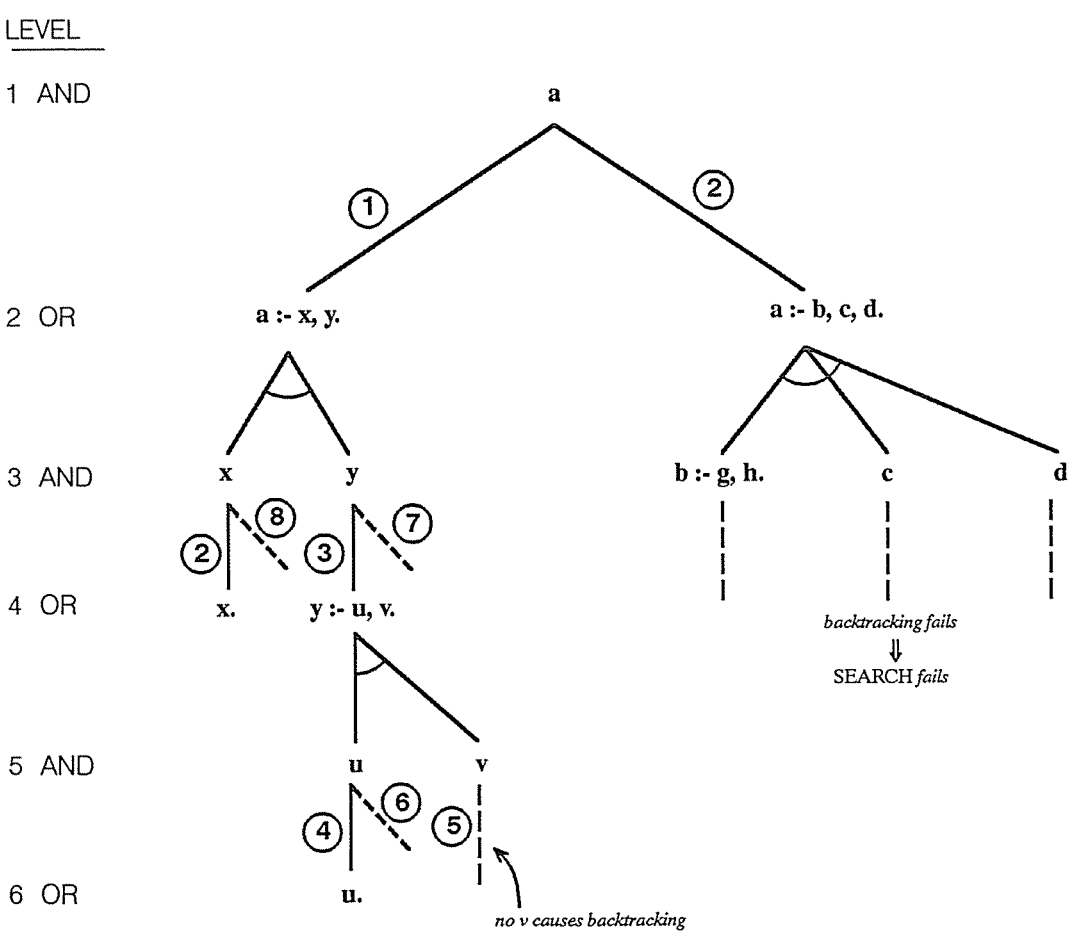
Iterative SEARCH

SEARCH (*-f*)

1. Set *goals* to point to **GOALS**; set **SOLVED** to empty; set *clauses* to point to **CLAUSES**.

(SEARCH) (*clauses -f*)

2. If **GOALS** is empty, then succeed: return (*- tf*)



Deduction tree of Prolog SEARCH

FIND-CLAUSE

3. If *clauses* is empty, go to 9.
4. Unify the first goal with the head of the first clause:
   unify(first(GOALS), first(first(*clauses*)))
   If unification succeeds, go to 6.
5. Remove the first clause from *clauses*:
   clauses ← tail(*clauses*) Go to 3.

REDUCE

6. Append *clauses* to SOLVED; append GOALS to SOLVED.
7. Set GOALS to the list formed by appending the clause body of the first clause in *clauses* to GOALS with unified goal removed:
   GOALS ← append((tail(first(*clauses*)), tail(GOALS))
8. Set *clauses* to point to CLAUSES. Go to 2.
9. If SOLVED is empty, fail: return (- *ff*)

BACKTRACK

10. Set GOALS to the first goals-list on SOLVED:
    GOALS ← first(SOLVED)
11. Set *clauses* to the first SOLVED-list clause pointer, incremented to the next clause:
    clauses ← tail(first(tail(SOLVED)))
12. Set SOLVED to SOLVED with the last state removed:
    SOLVED ← tail(tail(SOLVED))
    Go to 2.

NOTES:

1. The words *first* and *tail* are the LISP functions CAR and CDR. *First* returns the first item of a list. *Tail* returns the list that results from removing the first item from it.
2. When list *X* is appended to list *Y*, the result is a list containing the items of *X* followed by the items of *Y*.
3. Since list pointers, flags, etc. are passed on a stack, stack notation is used to indicate parameter passing. Stack items are indicated by italics.

SEARCH with iterative OR level and recursive AND level

SEARCH (*goals clauses* - *f*)

1. If *goals* is empty, then succeed: return (- *tf*)

FIND-CLAUSE (iterative)

2. If *clauses* is empty, then fail: return (- *ff*)
3. Unify the first goal with the head of the first clause:
   unify(first(*goals*), first(first(*clauses*)))
   If unification succeeds, go to 6.
4. Remove the first clause from *clauses*:
   clauses ← tail(*clauses*)
5. Go to 1.

`REDUCE` (recursive)

6. Set *newgoals* to the list formed by appending the clause body of the first clause in *clauses* to *goals* with unified goal removed:

   newgoals ← append((tail(first(*clauses*)), tail(*goals*))
7. Call `SEARCH` with (*newgoals* `CLAUSES` - *f*).
8. If `SEARCH` succeeds, return with true flag (- *tf*).
9. Go to 4.

Recursive `SEARCH`

`SEARCH` (*goals clauses* - *f*)

1. If *goals* is empty, then succeed: return (- *tf*)

`FIND-CLAUSE` (recursive)

2. If *clauses* is empty, then fail: return (- *ff*)
3. Unify the first goal with the head of the first clause:

   unify(first(*goals*), first(first(*clauses*)))

   If unification succeeds, go to 5.
4. Call `SEARCH` with (*goals* tail(*clauses*) - *f*).

`REDUCE` (recursive)

5. Set *newgoals* to the list formed by appending the clause body of the first clause in *clauses* to *goals* with unified goal removed:

   newgoals ← append((tail(first(*clauses*)), tail(*goals*))
6. Call `SEARCH` with (*newgoals* `CLAUSES` - *f*).
7. If `SEARCH` succeeds, return with true flag (- *tf*).
8. Go to 4.

NOTES:

1. The `SOLVED` list is not needed because state information is kept on the stack due to recursion. Backtracking occurs when `SEARCH` fails.

`SEARCH` written in Prolog

```
solve (goals)
solve ([ ]).
solve ([A|B]) :- solve (A), solve (B).
solve (A) :- clause ([A|B]), solve (B).
```

`SEARCH` written in Prolog using *solved* list

```
solve (goals, solved)
solve ([ ], [ ])
solve ([A|B], [U|V]) :- solve (A, U), solve (B, V).
solve (A, [A|X]) :- clause ([A|B]), solve (B, X).
```

NOTES:

1. [ ] is the symbol for the empty list in Prolog.
2. [*Y*|*Z*] is a list with *Y* as the first item and *Z* the rest of the list. Thus, first[*Y*|*Z*] → *Y* and tail[*Y*|*Z*] → [*Z*].
3. Symbol strings beginning with a capital letter are variables.