

# Describing Prolog by its interpretation and compilation

By Jacques Cohen

*Communications of ACM*- December 1985

Vol 28 number 12

The following illustrates through examples the main syntactic differences between the Marseilles (M) Prolog in Colmerauer's article (p. 1296) and the Edinburgh (E) Prolog used in this article.

<b>Variables</b>	(M) $x \ a \ x' \ x1^a$ (E) $X \ A \ Xprime \ X1^b$
<b>Constants</b>	(M) $123 \ abc^c$ (E) $123 \ abc$
<b>Rules</b>	(M) $a \rightarrow b \ c; \ a \rightarrow ;$ (E) $a \ :- \ b, \ c. \ a.$
<b>Lists</b>	(M) $a, b, x, nil \quad x, y$ (E) $[a, b, X] \quad [X Y]$

<sup>a</sup> Single letters followed by a prime or by digits.

<sup>b</sup> Identifiers starting with an uppercase letter.

<sup>c</sup> Integers or a sequence having more than two letters.

## DEFINITION

One concrete syntax for Prolog rules is given by

$\langle rule \rangle$	$::=$	$\langle clause \rangle .   \langle unit \ clause \rangle .$
$\langle clause \rangle$	$::=$	$\langle head \rangle \ :- \ \langle tail \rangle$
$\langle head \rangle$	$::=$	$\langle literal \rangle$
$\langle tail \rangle$	$::=$	$\langle literal \rangle \ \{, \langle literal \rangle \}$
$\langle unit \ clause \rangle$	$::=$	$\langle literal \rangle$

1.  $a :- b, c, d.$
2.  $a :- e, f.$
3.  $b :- f.$
4.  $e.$
5.  $f.$
6.  $a :- f.$

### **Boolean Semantics**

*a is true if b and c and d are true*

OR

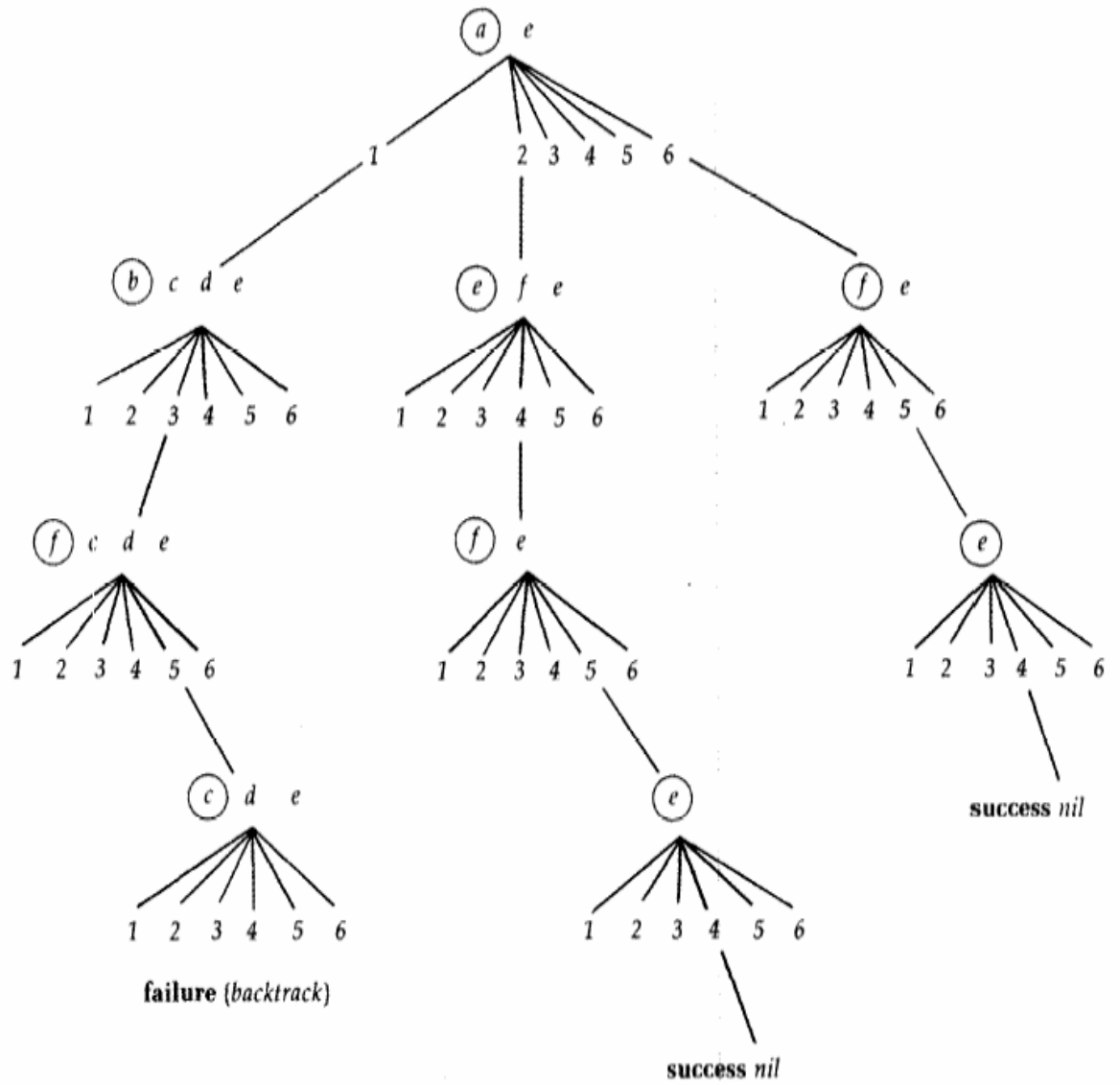
$b \wedge c \wedge d \rightarrow a.$

### **Procedural Semantics**

*goal a can be satisfied if goals b, c, and d can be satisfied.*

solution 1:  $a, e \Rightarrow e, f, e \Rightarrow f, e \Rightarrow e \Rightarrow nil$ ;

solution 2:  $a, e \Rightarrow f, e \Rightarrow e \Rightarrow nil$ .



**Rules:**

- |                    |              |
|--------------------|--------------|
| 1. $a :- b, c, d.$ | 4. $e.$      |
| 2. $a :- e, f.$    | 5. $f.$      |
| 3. $b :- f.$       | 6. $a :- f.$ |

**Query:**  $a, e.$

○ denotes the head of the list of goals.

```
procedure solve(L: pLIST);  
  begin local i: integer;  
    if L  $\neq$  nil  
      then  
        for i := 1 to n do  
          if match(head(Rule[i]), head(L))then  
            solve(append(tail(Rule[i]), tail(L)));  
          else write('yes')  
        end;  
  end;
```

**FIGURE 1.** An Initial Version of the Interpreter

```
function append(L1, L2: pLIST): pLIST;  
  if L1 = nil then append := L2  
    else append := cons(head(L1), append(tail(L1), L2));
```

```

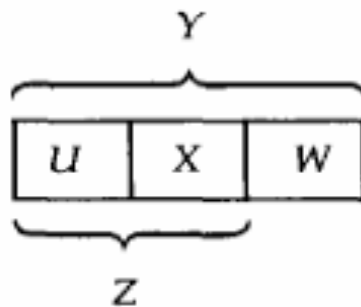
function append(L1, L2: pLIST; var L3: pLIST): Boolean;
  begin local H1, T1, T: pLIST;
    if L1 = nil then
      begin
        L3 := L2;
        append := true
      end
    else
      if {There exists an H1 and a T1 such that
        H1 = head(L1) and T1 = tail(L1)}
        then
          begin
            append := append(T1, L2, T);
            L3 := cons(H1, T)
          end
        else append := false
      end;
  end;

```



*sublist(X, Y) :- append(Z, W, Y), append(U, X, Z).*

where the variables represent the sublists indicated below:



An additional example is the bubblesort program credited to van Emden in [6]. The specification of two adjacent elements *A* and *B* in a list *L* is done by a call:

*append(—, [A, B|—], L)*

The underscore stands for a variable whose name is irrelevant to the computation, and the notation *[A, B|C]* stands for *cons(A, cons(B, C))*. (Note that the underscores correspond to different variables.) The rules to bubble-sort then become

*bsort(L, S) :- append(U, [A, B|X], L),  
                  B < A,  
                  append(U, [B, A|X], M),  
                  bsort(M, S),  
bsort(L, L).*

## Another Example: Game of Nim

```
us(X, Y):-          move(X, Y), not(them(Y, Z)).
them(X, Y):-        move(X, Y), not(us(Y, Z)).
move(X, Y):-        append(U,[X1|V], X),
                    takesome(X1, X2),
                    append(U,[X2|V], Y).

takesome(s(X), X).
takesome(s(X), Y):- takesome(X, Y).
```

FIGURE 3. A Program for Playing the Game of Nim

## UNIFICATION

Our previous definition of a *literal* is generalized to encompass labeled tree structures.

$\langle literal \rangle ::= \langle composite \rangle$   
 $\langle composite \rangle ::= \langle functor \rangle (\langle term \rangle \{, \langle term \rangle\}) |$   
 $\quad \langle functor \rangle$   
 $\langle functor \rangle ::= \langle lower\ case\ identifiers \rangle$   
 $\langle term \rangle ::= \langle constant \rangle | \langle variable \rangle | \langle composite \rangle$   
 $\langle constant \rangle ::= \langle integers\ and\ lower\ case\ identifiers \rangle$   
 $\langle variable \rangle ::= \langle identifiers\ starting\ with$   
 $\quad an\ upper\ case\ letter\ or\ \_ \rangle$

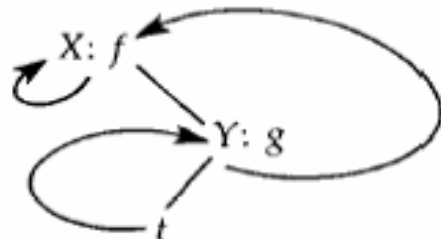
TABLE I.

Terms $1 \downarrow, 2 \rightarrow$	$\langle constant \rangle$ C2	$\langle variable \rangle$ X2	$\langle composite \rangle$ T2
$\langle constant \rangle$ C1	succeed if C1 = C2	succeed with X2 := C1	fail
$\langle variable \rangle$ X1	succeed with X1 := C2	succeed with X1 := X2	succeed with X1 := T2
$\langle composite \rangle$ T1	fail	succeed with X2 := T1	succeed if (1) T1 and T2 have the same functor and arity (2) the matching of corresponding children succeeds

## Unification using Infinite Trees

$eq(X, f(X, Y)), eq(Y, g(t(Y), X))$ .

produces the infinite tree



```
procedure solve(L, env: pLIST; level: integer);
  begin local i: integer; newenv: pLIST;
    if L ≠ nil
      then
        for i := 1 to n do
          begin
            newenv := unify(copy(head(Rule[i]), level + 1),
                           head(L), env);
            if newenv ≠ nil then
              solve(append(copy(tail(Rule[i]), level + 1), tail(L)),
                    newenv, level + 1)
            end
          else printenv(env)
        end
      end
  end;
```

FIGURE 4. A Final Version of the Interpreter

## Meta-Interpreter

```
solve(true).  
solve([Goal|Restgoal]) :- solve(Goal), solve(Restgoal).  
solve(Goal) :- clause(Goal, Tail), solve(Tail).
```

## A Meta-Interpreter defining Freeze

```
solve(true, Freezer, Freezer).  
solve([Goal|Restgoal], Freezer, NewFreezer):-  
    solve(Goal, Freezer, TempFreezer),  
    solve(Restgoal, TempFreezer,  
        NewFreezer).  
solve(Goal, Freezer, NewFreezer):-  
    clause(Goal, Tail),  
    defrost(Freezer, TempFreezer),  
    solve(Tail, TempFreezer, NewFreezer).  
solve(freeze(X, Goal), Freezer, [[X|Goal]|Freezer]):-  
    var(X).  
solve(freeze(X, Goal), Freezer, NewFreezer):-  
    nonvar(X),  
    solve(Goal, Freezer, NewFreezer).  
  
defrost([], []).  
defrost([[X|Goal]|Freezer], [[X|Goal]|NewFreezer]):-  
    var(X),  
    defrost(Freezer, NewFreezer).  
defrost([[X|Goal]|Freezer], NewFreezer):-  
    nonvar(X),  
    defrost(Freezer, TempFreezer),  
    solve(Goal, TempFreezer, NewFreezer).
```

**FIGURE 6.** Steps in the Unification Algorithm

