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.

Variables	(M)	*x*	*a*	*x'*	*x1*[a]
	(E)	*X*	*A*	*Xprime*	*X1*[b]
Constants	(M)	123	*abc*[c]		
	(E)	123	*abc*		
Rules	(M)	*a* → *b c*;		*a* → ;	
	(E)	*a* :- *b*, *c*.		*a*.	
Lists	(M)	*a.b.x.*nil		*x.y*	
	(E)	[*a*,*b*,*X*]		[X|Y]	

[a] Single letters followed by a prime or by digits.
[b] Identifiers starting with an uppercase letter.
[c] Integers or a sequence having more than two letters.

DEFINITION

One concrete syntax for Prolog rules is given by

```
⟨rule⟩          ::= ⟨clause⟩.|⟨unit clause⟩.
⟨clause⟩        ::= ⟨head⟩ :- ⟨tail⟩
⟨head⟩          ::= ⟨literal⟩
⟨tail⟩          ::= ⟨literal⟩ {,⟨literal⟩}
⟨unit clause⟩   ::= ⟨literal⟩
```

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

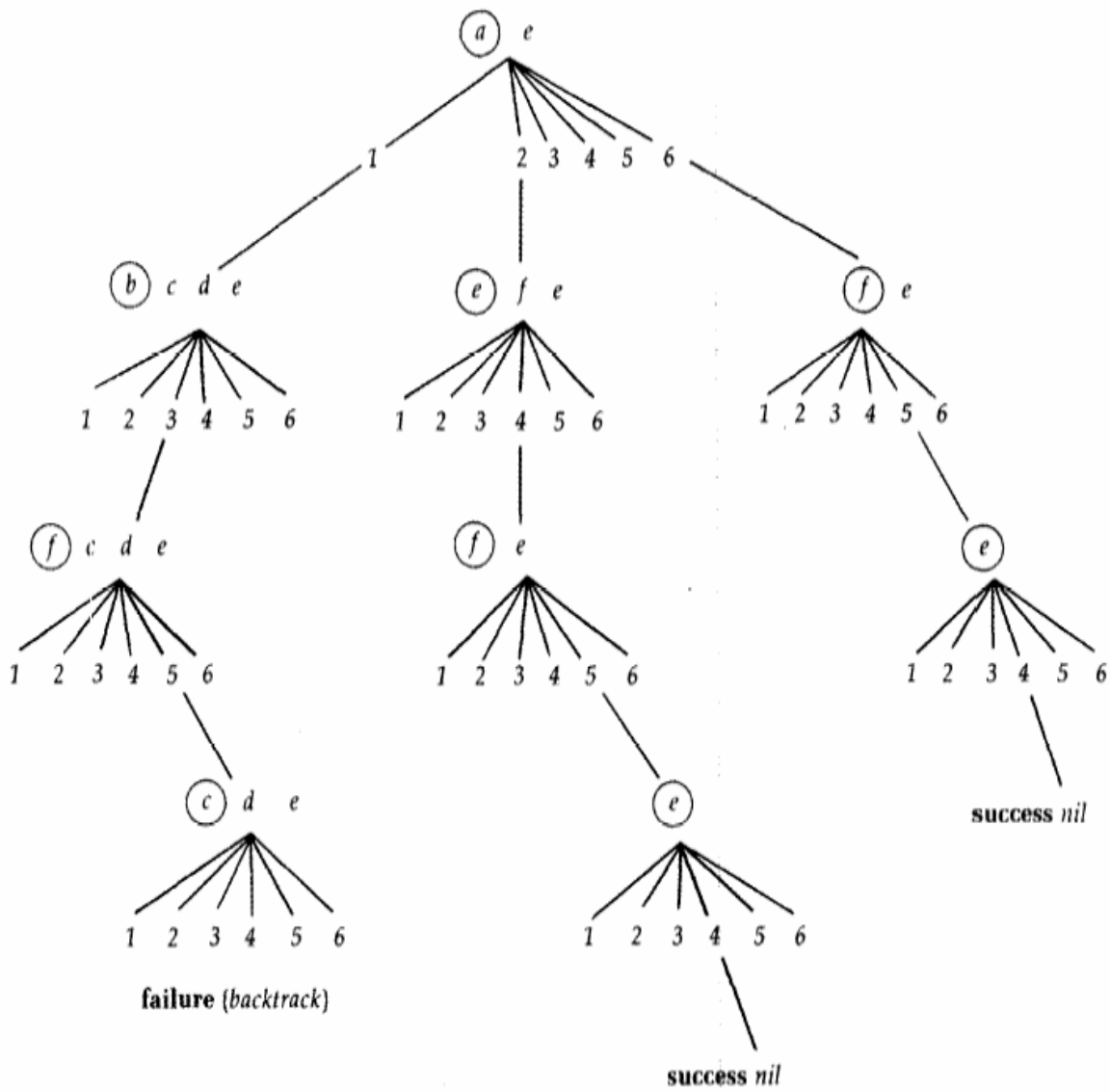
$b \wedge c \wedge d \rightarrow a.$

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.
2. a :- e, f.
3. b :- f.
4. e.
5. 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 ≠ 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;
```

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;
```

append(*L1*, *L2*, *L3*) **is true if** *L1* = *nil*
and
L3 = *L2*
otherwise
append(*L1*, *L2*, *L3*) **is true if** *L1* = *cons*(*H1*, *T1*)
and
append(*T1*, *L2*, *T*)

and
L3 = *cons*(*H1*, *T*)
otherwise *append* **is false.**

An equivalent program

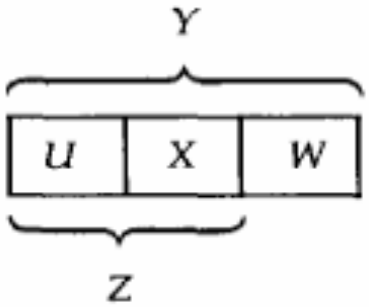
```
append(L1, L2, L3) :- L1 = nil, L3 = L2.
append(L1, L2, L3) :- L1 = cons(H1, T1),
                      append(T1, L2, T),
                      L3 = cons(H1, T).
```

Final version (actual Prolog)

```
append([ ], L2, L2).
append([H1 | T1], L2, [H1|T]) :- append(T1, L2, T).
```

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 bubblesort 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.

```
⟨literal⟩     ::= ⟨composite⟩
⟨composite⟩   ::= ⟨functor⟩ (⟨term⟩ {, ⟨term⟩})|
                  ⟨functor⟩
⟨functor⟩     ::= ⟨lower case identifiers⟩
⟨term⟩        ::= ⟨constant⟩|⟨variable⟩|⟨composite⟩
⟨constant⟩    ::= ⟨integers and lower case identifiers⟩
⟨variable⟩    ::= ⟨identifiers starting with
                  an upper case letter or _⟩
```

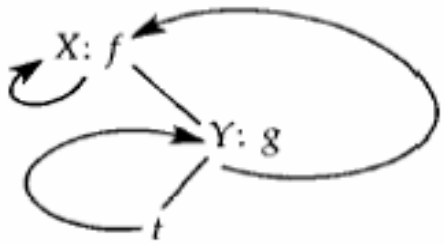
TABLE I.

Terms 1↓, 2→	⟨*constant*⟩ C2	⟨*variable*⟩ X2	⟨*composite*⟩ T2
⟨*constant*⟩ C1	*succeed if* C1 = C2	*succeed with* X2 := C1	*fail*
⟨*variable*⟩ X1	*succeed with* X1 := C2	*succeed with* X1 := X2	*succeed with* X1 := T2
⟨*composite*⟩ 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;
```

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, New Freezer).
defrost([[X|Goal]|Freezer], NewFreezer):-
                    nonvar(X),
                    defrost(Freezer, TempFreezer),
                    solve(Goal, TempFreezer, NewFreezer).
```

FIGURE 6. Steps in the Unification Algorithm

The predicate *Dif*

```
dif(X, Y) :- freeze(X, freeze(Y, different(X, Y))).
```

in which the built-in predicate *different*(X, Y) would test whether or not the bound variable X is different from the bound variable Y. Actually, the procedure *different* would have to be much more complex to achieve some of the generality of *dif* in Prolog II. Consider, for example, the query

```
dif(X, Y), X = f(a, B), Y = f(A, b).
```

Map Coloring

```
color(red).      color(white).      color(blue).
validcolors(node(N1, C1), node(N2, C2)) :- dif(C1, C2),
                                           color(C1),
                                           color(C2).
```