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.

<table>
<thead>
<tr>
<th>Variables</th>
<th>(M) x a x' x1*</th>
<th>(E) X A Xprime X1b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>(M) 123 abc^c</td>
<td>(E) 123 abc</td>
</tr>
<tr>
<td>Rules</td>
<td>(M) a → b c; a →</td>
<td>(E) a := b, c. a.</td>
</tr>
<tr>
<td>Lists</td>
<td>(M) a,b,x,nil</td>
<td>x,y</td>
</tr>
<tr>
<td></td>
<td>(E) [a,b,X]</td>
<td>[X</td>
</tr>
</tbody>
</table>

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

**DEFINITION**
One concrete syntax for Prolog rules is given by

\[
\begin{align*}
\langle rule \rangle & ::= \langle clause \rangle . \langle unit\ clause \rangle . \\
\langle clause \rangle & ::= \langle head \rangle ::= \langle tail \rangle \\
\langle head \rangle & ::= \langle literal \rangle \\
\langle tail \rangle & ::= \langle literal \rangle \{,\langle literal \rangle \} \\
\langle unit\ clause \rangle & ::= \langle literal \rangle
\end{align*}
\]
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

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

**Procedural Semantics**

goal $a$ can be satisfied if goals $b$, $c$, and $d$ can be satisfied.
solution 1:  $a, e \Rightarrow e, f, e \Rightarrow f, e \Rightarrow e \Rightarrow \text{nil};$

solution 2:  $a, e \Rightarrow f, e \Rightarrow e \Rightarrow \text{nil}.$
Rules:
1. \( a \leftarrow b, c, d. \)
2. \( a \leftarrow e, f. \)
3. \( b \leftarrow f. \)
4. \( e. \)
5. \( f. \)
6. \( a \leftarrow f. \)

Query: \( a, e. \)

\( \Box \) 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;
  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;
An equivalent program

\[
\begin{align*}
\text{append}(L_1, L_2, L_3) \text{ is true if } & \quad L_1 = \text{nil} \\
& \quad \text{and} \\
& \quad L_3 = L_2 \\
\text{otherwise} \\
\text{append}(L_1, L_2, L_3) \text{ is true if } & \quad L_1 = \text{cons}(H_1, T_1) \\
& \quad \text{and} \\
& \quad \text{append}(T_1, L_2, T) \\
\end{align*}
\]

\[ \text{and} \]
\[ L_3 = \text{cons}(H_1, T) \]

\text{otherwise append is false.}

Final version (actual Prolog)

\[
\begin{align*}
\text{append}(L_1, L_2, L_3) :- \ & \text{L}_1 = \text{nil}, \text{L}_3 = \text{L}_2. \\
\text{append}(L_1, L_2, L_3) :- \ & \text{L}_1 = \text{cons}(H_1, T_1), \\
& \text{append}(T_1, L_2, T), \\
& \text{L}_3 = \text{cons}(H_1, T). \\
\end{align*}
\]
\[ \text{sublist}(X, Y) :- \text{append}(Z, W, Y), \text{append}(U, X, Z). \]

where the variables represent the sublists indicated below:

\[
\begin{array}{c}
\text{Y} \\
\hline \\
U & X & W \\
\hline \\
\text{Z}
\end{array}
\]

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:

\[ \text{append}(_, [A, B|\_], L) \]

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

\[ \text{bsort}(L, S) :- \text{append}(U, [A, B|X], L), \]
\[ B < A, \]
\[ \text{append}(U, [B, A|X], M), \]
\[ \text{bsort}(M, S), \]
\[ \text{bsort}(L, L). \]
Another Example: Game of Nim

\begin{verbatim}
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).
\end{verbatim}

FIGURE 3. A Program for Playing the Game of Nim
UNIFICATION
Our previous definition of a \( \text{literal} \) is generalized to encompass labeled tree structures.

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

<table>
<thead>
<tr>
<th>Terms 1↓, 2→</th>
<th>\langle constant \rangle</th>
<th>\langle variable \rangle</th>
<th>\langle composite \rangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>\langle constant \rangle</td>
<td>succeed if C1 = C2</td>
<td>succeed with X2 := C1</td>
<td>fail</td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\langle variable \rangle</td>
<td>succeed with X1 := C2</td>
<td>succeed with X1 := X2</td>
<td>succeed with X1 := T2</td>
</tr>
<tr>
<td>X1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\langle composite \rangle</td>
<td>fail</td>
<td>succeed with X2 := T1</td>
<td>succeed if \begin{enumerate} \item T1 and T2 have the same functor and arity \item the matching of corresponding children succeeds \end{enumerate}</td>
</tr>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 \neq nil
            then
                for i := 1 to n do
                    begin
                        newenv := unify(copy(head(Rule[i]), level + 1),
                        head(L), env);
                        if newenv \neq 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 \textit{Dif}

\[
dif(X, Y) :- \text{freeze}(X, \text{freeze}(Y, \text{different}(X, Y))).
\]

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

\[
dif(X, Y), X = f(a, B), Y = f(A, b).
\]

Map Coloring

\[
\text{color(red)}, \text{color(white)}, \text{color(blue)}. \\
\text{validcolors(node(N1, C1), node(N2, C2)) :- dif(C1, C2),} \\
\quad \text{color(C1),} \\
\quad \text{color(C2).}
\]