PROLOG -
 the language
 and its implementation
 compared with Lisp

David Warren
University of Edinburgh

Luis Pereira & Fernando Pereira
National Civil Engineering Laboratory
Lisbon
Our Work

What?
Have implemented a Prolog compiler (and interpreter) for the DECSYSTEM 10 machine (written in Prolog).

Why?
......
The Prolog Language

- "programming in logic"
- developed by Colmerauer et al., University of Marseille
- noteworthy for:
  - fast, error-free programming
  - clear, readable, concise programs
Prolog dispenses with:

- goto
- do, for or while loops
- assignment
- references (pointers)

cf. pure Lisp
Viewpoint: Prolog as a generalisation of pure Lisp

- functions → general procedures
- lists → general tree structures
- function evaluation → procedure invocation
- constructors + selectors → pattern matching
- lambda calculus → classical logic (subset)
Elementary Data Objects ("terms")

- "atoms"
  
a  nil  tom  n1

- "integers"
  
0  -1  99999

- "variables"
  
X  Y2  Jim  C1
**Complex Data Objects ("terms")**

<table>
<thead>
<tr>
<th>&quot;functor&quot;</th>
<th>&quot;arguments&quot;</th>
<th>alternative form</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.e. record</td>
<td>i.e. fields of the record</td>
<td></td>
</tr>
<tr>
<td>point</td>
<td>(0, 10, -10)</td>
<td></td>
</tr>
<tr>
<td>pair</td>
<td>(romeo, juliet)</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>(a, b)</td>
<td>a + b</td>
</tr>
<tr>
<td>·</td>
<td>(Head, Tail)</td>
<td>Head · Tail</td>
</tr>
<tr>
<td>·</td>
<td>(1, (2, (3, nil)))</td>
<td>1 · 2 · 3 · nil</td>
</tr>
</tbody>
</table>
Procedure Call ("goal")

<table>
<thead>
<tr>
<th>&quot;predicate&quot;</th>
<th>&quot;arguments&quot;</th>
<th>alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>gives</td>
<td>(tom, apple, teacher)</td>
<td></td>
</tr>
<tr>
<td>reverse</td>
<td>((1, 2, 3, nil), List)</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>(X, Y)</td>
<td>X &lt; Y</td>
</tr>
</tbody>
</table>
Program Statements ("clauses")

- "non-unit clause"
  \[ P : - Q, R, S. \]
  \( P \) is true if \( Q, R \) and \( S \) are true.
  To satisfy goal \( P \), satisfy goals \( Q, R \) and \( S \).

- "unit clause"
  \[ P. \]
  \( P \) is true.
  Goal \( P \) is satisfied.

- "question"
  \[ ?- P. \]
  Is \( P \) true?
  Satisfy goal \( P \).
Variables in Clauses

NB. The "lexical scope" of a variable is limited to a single clause.

clause

bird(X):- crow(X).

some possible readings

Any X is a bird if X is a crow.
To find an X which is a bird,
find an X which is a crow.

helps(X,Y).

Everyone helps himself.

The goal of finding a person who helps X is satisfied by X himself.

?- employs(X,Y).

Does anyone employ himself?
Find an X who is self-employed.
Geographical Statistics Example

\[
\begin{align*}
\text{pop (china, 825).} & \quad \text{area (china, 3380).} \\
\text{pop (india, 586).} & \quad \text{area (india, 1139).} \\
\text{pop (ussr, 252).} & \quad \text{area (ussr, 8708).} \\
\text{pop (usa, 212).} & \quad \text{area (usa, 3609).}
\end{align*}
\]

\[
density(C, D) : \leftarrow \text{pop}(C, P), \text{area}(C, A), \text{D is } (P \times 1000)/A.
\]

\[
similar_densities(C_1, C_2) : - \\
\text{density}(C_1, D_1), \text{density}(C_2, D_2), \\
D_1 \geq D_2, \quad D_1 \times 20 < D_2 \times 21, \quad C_1 \neq C_2.
\]
Using the Geographical Statistics Program

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?- similardensities(france, china).</code></td>
<td>yes</td>
</tr>
<tr>
<td><code>?- density(france, X).</code></td>
<td>$X = 24.6$</td>
</tr>
<tr>
<td><code>?- similardensities(X, pakistan).</code></td>
<td>$X = \text{indonesia}$</td>
</tr>
<tr>
<td><code>?- similardensities(pakistan, X).</code></td>
<td>no</td>
</tr>
<tr>
<td><code>?- similardensities(X, Y).</code></td>
<td></td>
</tr>
</tbody>
</table>

$x =$ \text{indonesia} \quad y =$ \text{pakistan}

$x =$ \text{uk} \quad y =$ \text{w\_germany}

$x =$ \text{italy} \quad y =$ \text{philippines}

$x =$ \text{france} \quad y =$ \text{china}

The Declarative Semantics of Prolog

Prolog program

\[ \begin{align*}
P & : - Q, R. \\
P & : - R, S. \\
R & : - S. \\
S & :
\end{align*} \]

\[ \begin{align*}
\vdots \\
P_1. \\
P_2. \\
P_3. \\
\vdots \\
R_1. \\
R_2. \\
\vdots \\
S_1. \\
S_2. \\
\vdots
\end{align*} \]
The Declarative Semantics, Precisely Stated

A goal is true if it is the head of some clause instance and each of the goals (if any) in the body of that clause instance is true.

where

An instance of a clause (or term) is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable.
The Procedural Semantics of Prolog

- Execute a goal by matching* it against the head of a clause and then executing the clause's body.
- Execute goals in left-to-right order.
- Try clauses in top-to-bottom order.
- If a match can't be found, "backtrack".

* Matching ("unification") finds the most general common instance of goal and clause head.
A Conventional Program for the 'similardensities' procedure

Case: ?- similardensities (C1, C2).

for C1 from 1 to N do
  integer D1 := density (C1);
  for C2 from 1 to N do
    integer D2 := density (C2);
    if D1 ≥ D2 and D1×20 < D2×21
      and C1 ≠ C2
      then output (C1, C2)
  repeat
repeat
**Concatenating Lists**

**Lisp:**

\[
\text{append}[L1; L2] =
\]

\[
[\text{null}[L1] \rightarrow L2;
\]

\[
T \rightarrow \text{cons}[\text{car}[L1]; \text{append}[	ext{cdr}[L1]; L2]]]
\]

**Prolog:**

append(nil, L, L).

\[
\text{append}((X \cdot L1), L2, (X \cdot L3)) :-
\]

append(L1, L2, L3).
Different Uses of the 'append' Procedure

?- append((a\cdot b\cdot c\cdot \text{nil}), (d\cdot e\cdot \text{nil}), X).

?- append(X, Y, (a\cdot b\cdot c\cdot d\cdot e\cdot \text{nil})).

member(X, L) :- append(L1, (X\cdot L2), L).
Differentiating an Algebraic Expression

\[ d(U+V, X, DU+DV) :\]
\[ \quad d(U, X, DU), \; d(V, X, DV). \]

\[ d(U \times V, X, DU \times V + U \times DV) :\]
\[ \quad d(U, X, DU), \; d(V, X, DV). \]

\[ d(X, X, 1). \]

\[ d(C, X, 0) :\] atomic(C), C \neq X.
The Equivalent Differentiation Procedure in Lisp

```
DERIV (LAMBDA (E X))
  (COND ((ATOM E) (COND ((EQ E X) 1) (T 0)))
  ((EQ (CAR E) (QUOTE PLUS))
    (LIST (CAR E)
      (DERIV (CADR E) X) (DERIV (CADDR E) X)))
  ((EQ (CAR E) (QUOTE TIMES))
    (LIST (QUOTE PLUS)
      (LIST (CAR E) (CADDR E) (DERIV (CADR E) X))
      (LIST (CAR E) (CADR E) (DERIV (CADDR E) X)))))
```

cf. Weissman's Lisp 1.5 Primer
Special Characteristics of Prolog — Part I

- general record structures
  - no type restrictions
  - no declaration of record types
- pattern matching — replaces selectors + constructors
- more flexible procedures
  - multiple outputs
  - multi-purpose procedures
  - generate multiple alternative results
- backtracking $\equiv$ iteration
- "logical variable" $\equiv$ assignment + references
Special Characteristics of Prolog - Part II

- program and data expressed the same way
- declarative semantics in addition to the usual procedural semantics
- the procedural semantics is totally defined
Prolog supported by a practical interactive system

- Built-in procedures for
  - arithmetic
  - input-output
  - file handling
  - maintaining an in-core "database"
  - state saving

- Debugging aids
  - tracing
  - interrupts
  - on-line program amendment

- Extra operator for limiting the generation of alternatives
Some Applications of Prolog - large-scale programs

- natural language understanding         Colmerauer+al.
- algebraic symbol processing            Kanoui+Bergman
                                            Bundy+al.
- architectural design                   Markusz
- drug design applications               Darvas+Futo
- compiler writing                       Warren+al.
### Prolog Implementations

<table>
<thead>
<tr>
<th>When</th>
<th>Who</th>
<th>Where</th>
<th>What</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td>1972</td>
<td>Roussel</td>
<td>Marseille</td>
<td>interpreter</td>
<td>Algol-W</td>
</tr>
<tr>
<td>1974</td>
<td>Battani + Meloni</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Fortran</td>
</tr>
<tr>
<td>1975</td>
<td>Szeredi</td>
<td>Budapest</td>
<td>&quot;</td>
<td>CDL (Koster)</td>
</tr>
<tr>
<td>1976</td>
<td>Bruynooghe (Belgium)</td>
<td>&quot;</td>
<td>&quot;</td>
<td>Pascal</td>
</tr>
<tr>
<td>1976</td>
<td>Roberts</td>
<td>Waterloo</td>
<td>&quot;</td>
<td>360/370 assembler</td>
</tr>
<tr>
<td>1976</td>
<td>Warren + Pereira + Lisbon</td>
<td>Edinburgh → DEC10</td>
<td>compiler</td>
<td>Prolog</td>
</tr>
</tbody>
</table>
Innovations in Our Implementation

- Compilation

- Storage economy (with "structure sharing")
  - conventional stack mechanism to reclaim "local" variables
  - garbage collector for "global" variables
  - "mode declarations"

- Indexing of clauses
  (computed goto replaces sequence of tests)
Compiling the 'append' Procedure (first clause)

append (begin
    local variables L1, L2;
    global variables X, L3;
    \cdot (X, L1, L2) :-
        check term[1] is of type ";
        X := car(term[1]);
        L1 := cdr(term[1]);
        L2 := term[2];
        term[3] := cons(X, L3);
    append(L1, L2, L3).
    call append(L1, L2, L3);
    return;
end
Compiling the 'append' Procedure (second clause)

append( nil, L, L) begin

  temporary variable L;

  if term[1] is a variable
    then term[1] := nil
    else check term[1] is "nil";

  L := term[2];
  match (term[3], L);
  return

end
Comparative Performance Data
- time ratios for compiled code on DEC10

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>Lisp*</th>
<th>Pop-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>list concatenation</td>
<td>1</td>
<td>0.64</td>
<td>3.8</td>
</tr>
<tr>
<td>differentiation</td>
<td>1</td>
<td>1.3</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1.7</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2.6</td>
<td>5.4</td>
</tr>
<tr>
<td>geographical stats</td>
<td>1</td>
<td>-</td>
<td>1.6</td>
</tr>
</tbody>
</table>

* Stanford, + NOUVO option.
Conclusion I

Pattern matching is better than selectors+constructors.

- More readable.
- Faster!
  - procedure argument passing not just "red tape"
  - each component only selected once
  - all components selected in a single process
    - no reloading of index registers
    - no duplication of type checks
  - ("structure-sharing" gives faster "cons")
Conclusion II

In principle, Prolog supersedes Lisp...

- More readable
  - smaller program units
  - less complex (nested)
  - natural declarative reading

- General record structures replace lists
- General procedures replace simple functions
- Pure Prolog provides high-level substitutes for
  - iteration
  - assignment
  - references

- Prolog is practically as efficient as Lisp