Logical Programming with Prolog

Klaus Köhler
Munich University of Applied Sciences
FH - München
E-Mail: koehler@hm.edu
Home Page: http://www.cs.hm.edu/~koehler

pgp Key ID: 0xBE42584D, Algorithm: DSA/ELG, Size: 1024/1792
fingerprint: 8D67 19C6 70C7 14D0 E33D 42AA AA36 42F3 BE42 584D
Contents

1. History of Logic Programming
2. Introduction to Prolog
   Example: Symbolic Differentiation
3. Language Characteristics
4. Prolog Syntax
5. Predefined Predicates
   with interspersed examples
6. Backtracking
   Tracing
7. Predicate Logic and Horn Clauses
History of Logic Programming

- 1935 Logic foundations
  Gottlob Frege, Thoralf Skolem, Jacques Herbrand, Kurt Gödel

1972 Marseille PROLOG
  Alain Colmerauer

1972 Edinburgh PROLOG
  Robert Kowalski

- 1997 PROLOG IV

1980 Japanese "Fifth Generation Project"

- 1995 Development of a new hardware-OS-software architecture
  software based on PROLOG (not successful)
Introduction: Symbolic Differentiation

- **Define a knowledge base:** Example:
  - facts, assertions: \( \frac{dx}{dx} = 1 \)
  - rules: \( \frac{dC}{dx} = 0 \) if C is constant

- **Ask questions:**
  \( \frac{d3}{dx} = \text{What?} \)

  **Unification** of goal \( \frac{d3}{dx} = \text{What} \) with rule head \( \frac{dC}{dx} = 0 \) by
  instantiations:
  - variable C ↔ number 3,
  - variable X ↔ atom x,
  - variable What ↔ number 0

- **Prolog** (PROgramming in LOGic) knowledge base:
  - \( d(A,X,B) \) is a relation (predicate = facts or rules)
    with variables A, X, and B (arity 3), meaning \( \frac{dA}{dx}=B \)
  - \( d(X,X,1) \) is a fact, meaning \( \frac{dX}{dx}=1 \)
  - \( d(C,X,0) :\text{atomic}(C) \quad C \neq X \) is a rule, meaning
    \( \frac{dC}{dx}=0 \) if C is atomic and C \neq X

- **Prolog questions:**
  \( \text{?- } d(3,x,\text{What}) \).
Introduction: Symbolic Differentiation (cont.)

- **Extend the knowledge base:**
  
  
  rule: \[ \frac{d(U+V)}{dX} = A+B \quad \text{if} \quad \frac{dU}{dX}=A \quad \text{and} \quad \frac{dV}{dX}=B \]

  \[
  \downarrow \quad \downarrow \quad \downarrow
  \]

  **Prolog:**
  
  \[
  d(U+V, X, A+B) :- d(U, X, A) , d(V, X, B).
  \]

  **goal:**
  
  \[
  ?- d(x+1, x, What).
  \]

  \[
  \downarrow \quad \downarrow \quad \downarrow
  \]

  try to unify with:

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

  fails

  next try:

  \[
  d( C, X, 0) :- \text{atomic}(C) \ , \ C\neq X.
  \]

  success, leads to

  subgoals

  atomic(x+1) \quad \text{and} \quad x+1\neq x

  fails

  next try:

  \[
  d(U+V, X, A+B) :- d(U, X, A), d(V, X, B).
  \]

  success, leads to

  subgoals

  \[
  d(x, x, A) \quad \text{and} \quad d(1, x, B)
  \]

  \[
  \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow
  \]

  unify with:

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

  success

  and \[d(X, X, 1)\] fails

  try

  \[
  d(C, X, 0) :- \text{atomic}(C), C\neq X.
  \]

  success

  atomic(1), 1\neq x.

  success

result: unifications \[What=1+0\]
Prolog: Language Characteristics

- **Knowledge Base**: (declarative aspect)
  - **Facts**: \( d(X,X,1). \) /* head */
  - **Rules**: \( d(C,X,0) :- \text{atomic}(C) \), \( C\neq X \). /* head :- body */
  - **Goals**: \(?- d(3,x,\text{What}). \) /* body */

- **Unification (pattern matching)**
  - **Variables**: instantiation through unification (no assignment)
    - A (not instantiated) Variable matches any structure.
    - What matches 1 or \( A+B \) or ...
  - **Structure**: functor with optional parameter list e.g. \( d(3,x,\text{What}) \)
    - A structure matches another structure if
      - both have the same functor and
      - the corresponding arguments match
    - \( d(1+0,x,\text{What}) \) matches \( d(U+V,X,A+B) \) or \( d(X,Y,1) \) or ...

- **Asking Questions**: (procedural aspect)
  - Try to unify the goal with the head of a clause in the knowledge base (in the order of the entries) and instantiate variables.
    - clause = fact: present the variable instantiations (or just yes)
    - clause = rule: recursively try to unify the body (subgoals)
**Prolog: Syntax** (simplified)

A goal is a body.

All clauses with the same functor form a *Predicate (Relation).*

Operators are functors equipped with priority and associativity written in infix notation for binary operators prefix or postfix notation for monadic operators.
Prolog: List Examples

- **Is A member of the list L?**
  
  \[
  \begin{align*}
  \text{member}(A, [A|\_]). \\
  \text{member}(A, [\_|T]) & :\text{member}(A, T).
  \end{align*}
  \]

  **Usage:**
  
  \[- \text{member}(x, [y,x,z]). \\
  \text{- member}(E, [2,3,4]), \text{print}(E), \text{fail}.
  \]

- **Last element of a list**
  
  \[
  \begin{align*}
  \text{last}([X],X). \\
  \text{last}([H|T],L) & :\text{last}(T,L).
  \end{align*}
  \]

  **Usage:**
  
  \[- \text{last}([1,2,3],L). \\
  \text{- last}([1,2,3],3). \\
  \text{- last}(L,3).
  \]
Prolog: Predefined Predicates/Operators

- **General Predicates:**
  - `true`: always succeeds
  - `fail`: always fails
  - `not goal`: succeeds iff goal fails
  - `goal1, goal2`: logical and

- **Term Comparators:**
  - `term1 = term2`: unify term₁ with term₂ if possible
  - `term₁ \= term₂`: Is unification of term₁ with term₂ impossible?
  - `term₁ == term₂`: Are term₁ and term₂ identical, i.e. instances of the same term?
  - `term₁ \== term₂`: Are term₁ and term₂ not identical?
Prolog: Predefined Predicates (cont.)

- **Arithmetic Predicates und Operators:**
  - `+`, `-`, `*`, `/`, `mod` combine arithmetic expressions
  - `var is expr` calculate value of `expr` and unify it with `var`

- **Arithmetic Comparators:**
  - `=:=`, `=\ne`, `<`, `\le`, `>`, `\ge` compare arithmetic expressions consisting of numerical constants, instantiated variables, and arithmetic operators

- **Type Tests:**
  - `var(term)` Is term a free (not instantiated) variable? (term may be unified with another variable)
  - `integer(term)` Is term (instantiated with) an integer?
  - `atom(term)` Is term (instantiated with) an atom (= non-numeric constant)?
  - `atomic(term)` Is term (instantiated with) an atom or an integer?
Prolog: Arithmetic Examples

- nth element of a list
  
  \[\text{nth}(1, [H | _], H).\]
  
  \[\text{nth}(N, [_ | T], E) :- \text{integer}(N),\]
  \[N > 1, N1 \text{ is } N - 1,\]
  \[\text{nth}(N1, T, E).\]

  Usage:
  
  ?- \text{nth}(2, [1, 2, 3], E).
  ?- \text{nth}(2, \text{List}, 3).

- Length of a list
  
  \[\text{length}([], 0).\]
  
  \[\text{length}([H | T], L) :- \text{length}(T, TL), L \text{ is } TL + 1.\]

  Usage:
  
  ?- \text{length}([1, 2, 3], L).
  ?- \text{length}([1, 2, 3], 3).
  ?- \text{length}(L, 3). \% error when first solution is rejected
Prolog: Arithmetic Examples (cont.)

- Sum of the integer elements of a list
  
  \[
  \text{sum([],0).}
  \]
  
  \[
  \text{sum([H|T],L) :- integer(H), sum(T,TL), L is TL + H.}
  \]

  **Usage:**
  
  \[
  ?- \text{sum([1,2,3],S).}
  \]
  \[
  ?- \text{sum([1,2,3],3).}
  \]
  \[
  ?- \text{sum(L,3).}
  \]
Prolog: Predefined Predicates (cont.)

- **Structure of Terms:**
  - `functor(struct,f,a)` unify struct with a structure with functor f and arity a
  - `struct =~. list` unify list with a list `[fun|arg]`
    where the list head fun is the functor
    and the list tail arg is the argument list of struct

- **Example: Symbolic differentiation**
  ```prolog
  const(U,U) :- !, fail. % ensure that const(f(x),f(x)) fails
  const(C,X) :- atomic(C), C \= X.
  const(C,X) :- C=..[Op,Arg],const(Arg,X).
      % Op = FunctId/monadic operator
  const(C,X) :- C=..[_,U,V] ,const(U,X),const(V,X).
      % _ = dyadic operator
  
  ?- functor(S,const,2).
  ```
Prolog: Predefined Predicates (cont.)

- Access to and Modification of the Knowledge Base:
  - !: cut: stop redoing unification of this goal or its subgoals
  - asserta(clause): add clause to knowledge base at front of relation
  - assertz(clause): add clause to knowledge base at back of relation
  - retract(clause): retract first entry of knowledge base unified with clause
  - abolish(fun,a): retract whole predicate with functor fun and arity a
  - consult(file): read file and append its causes to the knowledge base or [file]
  - and immediately execute goals starting with ?- or :-
  - reconsult(file) or [-file]: read file and insert/override its causes
  - op(prio, ass, fun): define operator fun with priority prio, associativity ass
    - fx, fy: prefix x \leq term with operator priority higher than fun
    - xf, yf: postfix y \leq term with equal/lower operator priority
    - xfx, yfx, yfx: infix xfy: left associative, yfx: right associative

- Examples:
  - ?- consult(symdiff).
  - ?- op(250, yfx, ^). % define power operator upon loading
  - ?- assert(second([_,S|_],S)).
  - ?- abolish(second/2).
Prolog: Predefined Predicates (cont.)

- Input/Output Predicates:
  - `write(term)` output term to current stream
  - `read(term)` read term from current stream
  - `see(file)` make file current input stream
  - `seen` close current input stream, return to default stream `user`
  - `seeing(term)` unify term with current input stream
  - `tell(file)` make file current output stream
  - `told` close current output stream, return to default stream `user`
  - `telling(term)` unify term with current output stream

![Prolog I/O-Model](image)
Prolog: Exercises

Grammar for infix expressions
Expression ::= Term {(+|-) Term}
Term ::= Factor {(*|/) Factor}
Factor ::= SimpleFactor {^ SimpleFactor}
SimpleFactor ::= [+|-] (Symbol|Number|[FunctId] (Expression))
FunctId ::= sin | cos | exp | log

Examples:
pi
x
13
- 13
(x)
log(x)
x + 13 * + x / (- 13)
pi ^ - sin(x)
+ ((pi ^ sin(x) / (x + - 13 * x)))
Differentiation rules for infix expressions

Expression: \( (u \pm v)' = u' \pm v' \)

Term: \( (u \times v)' = (u' \times v) + (u \times v') \)
\( (u / v)' = ((u' \times v) - (u \times v')) / (v ^ 2) \)

Factor: \( (u ^ v)' = (\exp(\log(u^v)))' = (\exp(v'\log(u)))' = (\exp(v'\log(u))) \times (v'\log(u))' \)
\( = (u ^ v) \times (v \times \log(u))' \)

SimpleFactor: \( (+ u)' = u' \)
\( (- u)' = - u' \)
Symbol' = 1 if Symbol = "x" and 0 else
Number' = 0
\( (u)' = u' \)
\( \sin(u)' = \cos(u) \times u' \)
\( \cos(u)' = -\sin(u) \times u' \)
\( \exp(u)' = \exp(u) \times u' \)
\( \log(u)' = u' / u \)
Prolog: Exercises (cont.)

Prolog program for symbolic differentiation (without sin, cos, exp)

?- op(350, xfy, ^). % define power operator
% x' = 1
d(X,X,1) :- !.
% c' = 0
d(C,X,0) :- atomic(C).
d(-U,X,-A) :- d(U,X,A).
d(+U,X,A) :- d(U,X,A).
d(U+V,X,A+B) :- d(U,X,A), d(V,X,B).
d(U-V,X,A-B) :- d(U,X,A), d(V,X,B).
d(U*V,X,B*A+V) :- d(U,X,A), d(V,X,B).
d(U/V,X,(A*V-U*B)/V^2) :- d(U,X,A), d(V,X,B).
d(U^C,X,C*A*U^(C-1)) :- atomic(C), C \= X, d(U,X,A).
d(log(U),X,A*U^(-1)) :- d(U,X,A).

Usage:
?- d(x+1,x,X).
  X = 1 + 0 ; ← ; or no reject the answer
  no
Prolog: Exercises (cont.)

1. Enhance the program so that all expressions can be derived.
2. Simplify the resulting expressions through adding the differentiation rule 
\((C^*U)' = C^*U'\) für atomic(C) und \(C \neq X\)

3. What happens if you forget the cut after the simpler rule in 1. and 2.?

**Remark:**
Simplification of expression is much more complicated than derivation.

**Example 1:** derivation of monomials: 
\((u ^ c)' = c * u ^ (c-1) * u'\) for integer c

- \(d(U^C,X,C^*A*U^C_1)) :- integer(C),!,C_1 \text{ is } C-1,d(U,X,A).\)

**Example 2:** derivation of constant expressions: 
\(c' = 0\)

- replace atomic(C) by const(C,X) defined by 
  - const(U,U) :- !, fail. % ensure that const(f(x),f(x)) fails
  - const(C,X) :- atomic(C), C \neq X.
  - const(C,X) :- C=..[Op,Arg],const(Arg,X). % monadic
  - const(C,X) :- C=..[_U,V] ,const(U,X),const(V,X). % dyadic
4. Assume you use a cut instead of C\=X. What yield the goals
   \- d(x,x,1).
   \- d(x,x,0).

5. Can you use the derivation program for integration?
   Check the following goals using \(\text{spy}(d)\):  
   \- d(Integral,x,1).
   \- d(Integral,x,-1).  \% not: d(Integral,x,-1).
   \- d(Integral,x,1 + 0).
   \- d(Integral,x,x).

Prolog Lab Exercises

We use the freeware Prolog interpreter


Invoke the Prolog-Interpreter from the subdirectory that contains your source programs or make it your current directory.

Load your source through

\texttt{consult(file).} \ or \ [file]. \ % same as \texttt{reconsult} in SWI-Prolog

or

\texttt{reconsult(file).} \ or \ [- file].

Leave the Prolog interpreter with

\texttt{halt}.

Exercises

Develop the following Prolog predicates and send your programs via email to your lecturer.

1. Write a predicate \texttt{myLast} that finds the last element in a list.

2. Write a predicate that determines the dot-product of two vectors \(a = (a_1, a_2, \ldots, a_n)\) und \(b = (b_1, b_2, \ldots, b_n)\) represented by lists.

3. Define a predicate \texttt{myAppend} that concatenates two lists.

4. Define a predicate \texttt{reverse} that inverts the order of the elements of a list, e.g.

\[\text{reverse}([1, 2, 3], R) \Rightarrow R = [3, 2, 1].\]
Prolog: Backtracking

- **Flow control**
  - unification through pattern matching
  - backtracking when unification fails, e.g. by predicate fail
  - ! (cut) freezes region of subgoals and excludes it from backtracking

- **Cut Example: Minimum of the integer elements of a list**
  
  ```prolog
  min([X], X) :- !.
  min([H|T], M) :- integer(H), min(T, M), H>M, !.
  min([H|_], H). % anonymous variable _
  ```

  **Usage:**
  
  ```prolog
  ?- min([1,2,3], L).
  ?- min([1,2,3], 3).
  ?- min(L, 3).
  ```
Prolog: Backtracking Examples

- Connections in a directed graph
  
  `direct(koeln, berlin).`
  
  `direct(berlin, frankfurt).`
  
  `flight(A, B) :- direct(A, B).`
  
  `flight(A, B) :- direct(A, Z), flight(Z, B).`

Usage:

```prolog
?- print(koeln), print(' -> '),
   flight(koeln, To), print(To), print(', '),
   fail.    % force backtracking
?- flight(From, To),
   print(From), print(' -> '), print(To),
   print(', '), fail.
```


Prolog: Backtracking Examples (cont.)

Example

- knowledge base:
  - person(adam).
  - person(X) :- father(X,Y), person(Y).
    father(kain,adam).
- goal:
  - person(M).
- interpretation:
  - match goal $\text{person}(M)$ with the heads of all clauses (same functor and arity) and instantiate corresponding variables:
    \[ \text{person}(M) \leftrightarrow \text{person(adam)} \Rightarrow M \leftrightarrow \text{adam} \]
  - mark the matching point in the knowledge base
  - fact $\Rightarrow$ output instantiations of variables: $M=\text{adam}$
  - rule $\Rightarrow$ unify subgoals in given order
  - unification fails, e.g. by not accepting the result (enter ; or no)
    $\Rightarrow$ unbind variables, redo unification starting at last matching point
    \[ \text{person}(M) \leftrightarrow \text{person}(X) \Rightarrow M \leftrightarrow X \]
    \[ \Rightarrow \text{subgoals: father}(X,Y) \text{ and } \text{person}(Y). \]
  - match subgoal $\text{father}(X,Y) \leftrightarrow \text{father(kain,adam)} \Rightarrow M=kain, Y=\text{adam}$
  - match subgoal $\text{person(adam)} \leftrightarrow \text{person(adam)}$
  - output $M=kain$
Tracing

- Trace protocol
  ```prolog
  ?- person(M).
  CALL: person(M)
  EXIT: person(adam)
  M = adam;
  REDO: person(M)
  CALL: father(M, Y)
  EXIT: father(kain, adam)
  CALL: person(adam)
  EXIT: person(adam)
  M = kain;
  REDO: person(adam)
  CALL: father(adam, Y)
  FAIL: father(adam, Y)
  no
  ?-
  ```

- Knowledge base
  ```prolog
  person(adam).
  person(X) :- father(X, Y), person(Y).
  father(kain, adam).
  ```

---

Box Model

Predicate

CALL → EXIT

FAIL → no success

SUCCESS → REDO

K. Köhler: Concepts of Modern Programming Languages
**Prolog Exercises: procedural ↔ declarative**

- **Principle of relational programming:**
  - describe declarative knowledge (what)
  - leave procedural aspects (how) to the interpreter

- **Prolog: declarative and procedural aspects**
  - program can be declaratively correct but procedurally wrong

- **Example:**
  
  ```prolog
  person(X) :- person(Y), father(X,Y). % left recursion
  person(adam).
  father(kain, adam).
  
  Usage:
  ?- person(M). % infinite recursion
  ```

- **Procedural aspects of Prolog:**
  - order of clauses and subgoals in knowledge base
  - cut (exclude goals from backtracking)
  - dynamic modification of the knowledge base

- **Programming hint:** Concentrate on the declarative aspect, but
  - clause order: simple clauses (e.g. facts) first, recursive rules last
  - subgoal order: simple subgoals first, avoid left recursion
Prolog Exercises: procedural ↔ declarative (cont.)

1. **Order of clauses and subgoals:**
   Test the following knowledge bases (use `spy`):
   (1) `person(X) :- person(Y), father(X,Y).
       person(adam).
       father(kain,adam).
   (2) `person(X) :- father(X,Y), person(Y).
       person(adam).
       father(kain,adam).
   (3) `person(adam).
       person(X) :- person(Y), father(X,Y).
       father(kain,adam).

   - What are the results of the goal
     `?- person(M).`
   - Which programs terminate?
   - What happens after rejecting the result?

(3) Trace protocol: p ≡ person, f ≡ father, a ≡ adam, k ≡ kain

```
1  p(a) ==> M=adam
2   p(Y),
21   p(a)  ->
22  p(Y1),
221  p(a)  ->
222  p(Y2),
2221  p(a)  ->
2222  p(Y3),
22221  p(a)  ->
```

```
f(M,Y)
f(M,a)
f(k,a) ==> M=kain
f(k,a)  ->  f(M,k) fail
f(k,a)  ->  f(M,k) fail
f(Y,k) fail
f(k,a) ->  f(Y1,k) fail
```


2. Negation through cut; fail:
   \[ \text{not}(P) :- P, \!, \text{fail}. \]
   \[ \text{not}(P); \]
   - **Closed World Assumption:**
     - Every true proposition can be derived from the knowledge base.
     - \( \Rightarrow \) A proposition \( P \) is false if it cannot be derived from the knowledge base.
   - **CWA in combination with the procedural aspects of Prolog** can lead to unexpected/contradictory results:
     \[
     r(a).
     q(b).
     p(X) :- \text{not}(r(X)).
     \]
     What are the results of the declaratively equivalent goals
     \[
     ?- q(X), p(X). \text{ and } ?- p(X), q(X).
     \]
Predicate Logic and Horn Clauses

- **Predicate logic**
  Formulas of 1. level predicate logic consist of
  - proposition logic formulas, e.g. \( A \land B \to C \)
  - functions and relations (predicates) among variables, e.g. member(X,L)
  - quantifiers \( \forall, \exists \) for the variables, e.g. \( \forall A \forall B: (B \lor \neg A) \leftrightarrow (A \to B) \)

Represent predicate logics formulas as clauses:
- bind them by existence quantifier (existential closure)
- replace \( \exists \) with \( \forall \) by using Skolemization and \( \exists x: P(x) \iff \neg(\forall x: \neg P(x)) \)
- rewrite the formulas in *conjunctive normal form*:
  \[
  \forall x_1 \ldots \forall x_k: C_1 \land \ldots \land C_r \iff P(x_1, \ldots, x_k)
  \]
- Use \( (\neg B \lor A) \leftrightarrow (B \to A) \) or \( (\neg B \lor A) \leftrightarrow (A \leftarrow B) \) to replace
  \[
  C_i = L_1 \lor \ldots \lor L_m = \neg p_1(x_1, \ldots, x_k) \lor \ldots \lor \neg p_n(x_1, \ldots, x_k) \lor p_{n+1}(x_1, \ldots, x_k) \lor \ldots \lor p_m(x_1, \ldots, x_k)
  \]
  with \( p_{n+1}(x_1, \ldots, x_k) \lor \ldots \lor p_m(x_1, \ldots, x_k) \iff p_1(x_1, \ldots, x_k) \land \ldots \land p_n(x_1, \ldots, x_k) \)

- **Horn clauses**
  Restrict the formulas to only one non-negated atomic formula \( p_1(x_1, \ldots, x_k) \):
  \[
  p_1(x_1, \ldots, x_k) \iff p_1(x_1, \ldots, x_k) \land \ldots \land p_n(x_1, \ldots, x_k)
  \]

- **Prolog = Horn clauses with syntactic sugar**
  \[
  p(X1, \ldots, Xk) :- p1(X1, \ldots, Xk), \ldots, pn(X1, \ldots, Xk)
  \]

*Skolemization* of depending existence quantifiers: e.g. \( \forall x \exists y: P(x,y) \)
replaced by \( \forall x: P(x,f(x)) \) with a new function \( f \).
Horn Clauses and Closed World Assumption

- **OR-operator**
  - not available in Horn clauses ⇒ not necessary in Prolog
  - OR-operator ; available in Prolog ⇒ expressiveness, comfort
  - C :- A ; B. is equivalent to C :- A.  C :- B. since
    
    \[
    C \leftarrow A \lor B \iff \neg(A \lor B) \lor C
    \iff (\neg A \land \neg B) \lor C \iff (\neg A \lor C) \land (\neg B \lor C) \iff (C \leftarrow A) \land (C \leftarrow B)
    \]

- **Resolution**
  - Predicate logic is not decidable ⇒ no resolution algorithm possible
  - Horn clause logic is decidable ⇒ resolution algorithm available

- **CWA inconsistent with predicate logic**
  - CWA: If C cannot be inferred from the knowledge base, then ¬C holds.
  - Example:
    A ∨ B neither implies A nor B
    Therefore due to CWA: ¬A and ¬B, which implies ¬(A ∨ B)
    Contradiction!
