Features

- Programming using logic.
- Symbolic computation.
- Backtracking for control.
- Pattern-matching used for both data access and data construction.
- Logical variables.
- Arithmetic not logical.

Brief History

- Originated from research in *automated theorem proving*.
- Idea of *logic programming* first conceived by a group around Alain Colmerauer in France.
- First efficient (“compiled”) implementation by David H. D. Warren in Scotland.
- Chosen by the Japanese as the primary language for their *5th Generation Computer Project* in the 80s. Did not really go anywhere, but did popularize Prolog.
- Has currently been generalized to *constraint programming languages*.

Prolog Data

Prolog data entities are referred to as *terms*. *Terms* consists of:

- Numbers; integers, floating-point numbers.
- Atoms: identifier starting with lower-case letters; a sequence of special characters; any string withing single quotes `'`.

Prolog Data Continued

- Variables: identifiers starting with upper-case letters or underscores. A anonymous variable is denoted using simply a _; each occurrence of a anonymous variable can stand for different entities.
- If `f` meets the syntactic restrictions for a atom, and T_1, T_2, ..., T_n are terms, then so is `f(`T_1`,` T_2`,` ... T_n`)`. This sort of term is referred to as a *structure* with functor `f` and arity N. The functor and arity are often denoted together as `f/`N.

Syntactic Sugar

- Prolog supports user-defined *distfix* operators; this covers commonly used operators like arithmetic operators. Terms using operators are syntactic sugar for the corresponding structure; i.e., `1 + 2` is equivalent to `+(1, 2)`.
- The integer corresponding to the character code for character *c* can be be denoted as `0'`*c*.

Syntactic Sugar: Lists

- Pairs which are structures built using functor `./2` are denoted within square-brackets with the `|` operator; i.e., `.(a, b)` is denoted as `[a|b]`.
- Lists are defined in the usual way using pairs. The empty list is denoted as `[]`. A list of elements `[a, b, c]` is equivalent to `.(a, .(b, .(c, [])))`. A list with head `X` and tail `Xs` is denoted as `[X|Xs]`.
- A string within double quotes denotes the list of integers corresponding to the character codes of the characters in the string. For example, `"abc"` is syntactic sugar for `[97, 98, 99]`.

Pattern Matching

Almost all computation in Prolog is done using a general form of pattern matching termed *unification*.

- Unifying two terms finds a *most general substitution* for the variables in the two terms such that applying the substitutions makes the two terms *identical*.
- Unification is done implicitly but can be made explicit using the $=$ operator.

Pattern Matching Examples

```
?- X = a.
X = a ;
No
?- f(X, a) = f(a, Y).
X = a
Y = a ;
No
?- f(X, a) = f(a, X).
X = a ;
No
?- f(X, a) = f(b, X).
No
```

Pattern Matching Examples Continued

```
?- .(a, b) = [X|Y].
X = a
Y = b ;
No
?- [a, b, c] = [X|Y].
X = a
Y = [b, c] ;
No
?- [a, b, c] = [X, Y, Z].
X = a
Y = b
Z = c ;
No
```

Pattern Matching Examples Continued

```
?- [X, [a, b], X|Z] = [a, [X|Z], a, b].
X = a
Z = [b] ;
No
?-
```

Prolog Program

A Prolog program consists of a set of *facts* and *rules*.

- A *fact* consisting of a single "predicate" is unconditionally true.
- A *rule* $P :- Q$ says that **head** predicate P is true if Q is true, where the **body** Q can consists of multiple predicates separated by , denoting logical-and.

Ancestor Program

Example ancestors.pl:

```
%Set of facts about tom's genealogy.
mother(tom, jill).
mother(jill, mary).
mother(bill, marge).

father(tom, bill).
father(jill, frank).
father(bill, harry).

%General rule: A parent is a mother or a father.
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

%A ancestor is a parent or the ancestor of a parent
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).
```

Ancestor Log

```
?- ['ancestors.pl'].
Yes
?- mother(tom, jill).
Yes
?- mother(tom, mary).
No
?- mother(tom, X).
X = jill ;
No
?- mother(X, jill).
X = tom ;
No
```

Ancestor Log Continued

```
?- mother(X, mary).
X = jill ;
No
?- parent(tom, X).
X = jill ;
X = bill ;
No
?- ancestor(bill, X).
X = marge ;
X = harry ;
No
```

Ancestor Log Continued

```
?- ancestor(tom, X).
X = jill ;
X = bill ;
X = mary ;
X = frank ;
X = marge ;
X = harry ;
No
?- ancestor(X, harry).
X = bill ;
X = tom ;
No
```

Ancestor Log Continued

```
?- ancestor(X, Y).
X = tom
Y = jill ;
X = jill
Y = mary ;
X = bill
Y = marge ;
X = tom
Y = bill ;
```

Ancestor Log Continued

```
X = jill
Y = frank ;
X = bill
Y = harry ;
X = tom
Y = mary ;
X = tom
Y = frank ;
X = tom
Y = marge ;
X = tom
Y = harry ;
No
?-
```

Multi-Way Predicates

As the previous example illustrates, `ancestor/2` can be used with multiple instantiation patterns and is not only used to find ancestors, but can also be used to find descendents. In simple cases like this, these sort of multi-way predicates are easy to write, but in large programs, predicates are often used only for fixed instantiation patterns.
A useful multi-way predicate is append to append two lists together:

```
xappend([], X, X).
xappend([X|Xs], Ys, [X|Zs]):-
  xappend(Xs, Ys, Zs).
```

Append Log

```
?- ['xappend.pl'].
% xappend.pl compiled 0.01 sec, 620 bytes
Yes
?- xappend([a, b], [1, 2, 3], Z).
Z = [a, b, 1, 2, 3] ;
No
?- xappend([a, b], X, [a, b, 1, 2, 3]).
X = [1, 2, 3] ;
No
?- xappend(X, [1, 2, 3], [a, b, 1, 2, 3]).
X = [a, b] ;
No
```

Append Log Continued

```
?- xappend(X, Y, [a, b, 1, 2, 3]).
X = []
Y = [a, b, 1, 2, 3] ;
X = [a]
Y = [b, 1, 2, 3] ;
X = [a, b]
Y = [1, 2, 3] ;
X = [a, b, 1]
Y = [2, 3] ;
X = [a, b, 1, 2]
Y = [3] ;
X = [a, b, 1, 2, 3]
Y = [] ;
No
```

Append Log Continued

```
?- xappend(X, Y, Z).
X = []
Y = _G148
Z = _G148 ;
X = [_G235]
Y = _G148
Z = [_G235|_G148] ;
X = [_G235, _G241]
Y = _G148
Z = [_G235, _G241|_G148] ;
```

Append Log Continued

```
X = [_G235, _G241, _G247]
Y = _G148
Z = [_G235, _G241, _G247|_G148] ;
X = [_G235, _G241, _G247, _G253]
Y = _G148
Z = [_G235, _G241, _G247, _G253|_G148]
Yes
?-
```

Naive Reverse

A naive $O(n^2)$ reverse:

```
nrev([], []).
nrev([X|Xs], Zs):-
   nrev(Xs, Ys),
   append(Ys, [X], Zs).
```

Naive Reverse Log

```
?- ['nrev.pl'].
% nrev.pl compiled 0.00 sec, 672 bytes
Yes
?- nrev([a, b, c], Z).
Z = [c, b, a] ;
No
?- nrev(X, [1, 2, 3]).
X = [3, 2, 1] ;
Action (h for help) ? a
abort
% Execution Aborted
```

Naive Reverse Log

```
?- nrev(X, Y).
X = []
Y = [] ;
X = [_G221]
Y = [_G221] ;
X = [_G221, _G224]
Y = [_G224, _G221] ;
X = [_G221, _G224, _G227]
Y = [_G227, _G224, _G221] ;
X = [_G221, _G224, _G227, _G230]
Y = [_G230, _G227, _G224, _G221]
Yes
?-
```

Non-Termination and Operational Semantics

As the `nrev/2` log illustrates, termination is not guaranteed.
Every Prolog program can be given 2 kinds of semantics:

Declarative Semantics: The program is a collection of logical statements (facts and rules) about the world.

Operational Semantics: A Prolog interpreter operates on a Prolog program to infer one or more goals from the program.

Declarative Semantics

A Prolog program is a collection of declarative statements about the domain of interest:

- Each fact is unconditionally true.
- Each rule is a logical **implication**; the **conjunction** of body predicates implies the head predicate.
- The declarative meaning of the program is the **conjunction** of the facts and rules in the program.
- Ideally, each fact or rule should be a true statement about the domain, independent of any other fact or rule in the program.
- The ordering of the facts/rules in a program should not affect the declarative meaning of the program.
- The program logically implies the truth of further facts, some subset of which can be computed using the operational semantics.

Operational Semantics

Operational Semantics: The operation of a Prolog systems given a goal *G* to satisfy is as follows:

1. The system finds the first rule or fact whose head matches *G* and recursively attempts to solve all goals in the body with the substitution matching the head to *G* applied to the body.
2. If the attempt to solve the body succeeds, then the goal *G* succeeds.
3. If the body attempt fails then it repeats with the next rule or fact matching *G*. If all such attempts fail, then the goal *G* fails.

The operation constitutes a depth-first search to satisfy goal *G*.

nrev Loop

Why does nrev loop for `nrev(X, [1])`?

```
nrev([], []).
nrev([X|Xs], Zs):-
   nrev(Xs, Ys),
   append(Ys, [X], Zs).
```

nrev Loop

```
?- nrev(X, [1]).
X = [1] ;
Action (h for help) ? a
abort
% Execution Aborted
?- trace.
Yes
[trace]  ?- nrev(X, [1]).
   Call: (8) nrev(_G279, [1]) ?
   Call: (9) nrev(_G333, _L207) ?
   Exit: (9) nrev([], []) ?
   Call: (9) lists:append([], [_G332], [1]) ?
   Exit: (9) lists:append([], [1], [1]) ?
   Exit: (8) nrev([1], [1]) ?
X = [1] ;
```

nrev Trace Continued

```
Redo: (9) nrev(_G333, _L207) ?
Call: (10) nrev(_G336, _L227) ?
Exit: (10) nrev([], []) ?
Call: (10) lists:append([], [_G335], _L207) ?
Exit: (10) lists:append([], [_G335], [_G335]) ?
Exit: (9) nrev([_G335], [_G335]) ?
Call: (9) lists:append([_G335], [_G332], [1]) ?
Call: (10) lists:append([], [_G332], []) ?
Fail: (10) lists:append([], [_G332], []) ?
Fail: (9) lists:append([_G335], [_G332], [1]) ?
```

nrev Trace Continued

```
   Redo: (10) nrev(_G336, _L227) ?
   Call: (11) nrev(_G339, _L240) ?
   Exit: (11) nrev([], []) ?
   Call: (11) lists:append([], [_G338], _L227) ?
   Exit: (11) lists:append([], [_G338], [_G338]) ?
   Exit: (10) nrev([_G338], [_G338]) ?
   Call: (10) lists:append([_G338], [_G335], _L207)
   Call: (11) lists:append([], [_G335], _G348) ?
   Exit: (11) lists:append([], [_G335], [_G335]) ?
   Exit: (10) lists:append([_G338], [_G335], [_G338
% Execution Aborted
?-
```

Accumulating Reverse

Reverse which uses accumulating parameters to get a $O(n)$ reverse:

```
rev(Xs, Zs):-
  rev(Xs, [], Zs).

rev([], Acc, Acc).
rev([X|Xs], Acc, Zs):-
  rev(Xs, [X|Acc], Zs).
```

Accumulating Reverse Log

```
?- ['rev.pl'].
% rev.pl compiled 0.00 sec, 756 bytes
Yes
?- rev([1, 2, 3], X).
X = [3, 2, 1] ;
No
?- rev(X, [a, b, Z]).
X = [_G156, b, a]
Z = _G156 ;
ERROR: Out of global stack
?- rev([a, b, X], Z).
X = _G153
Z = [_G153, b, a] ;
No
```

Accumulating Reverse Log Continued

```
?- rev(X, Y).
X = []
Y = [] ;
X = [_G212]
Y = [_G212] ;
X = [_G212, _G218]
Y = [_G218, _G212] ;
X = [_G212, _G218, _G224]
Y = [_G224, _G218, _G212] ;
X = [_G212, _G218, _G224, _G230]
Y = [_G230, _G224, _G218, _G212]
Yes
?-
```

List Member

Predicate xmember to check whether specified element unifies with some member of a list.

```
xmember(X, [X|_]).
xmember(X, [_|Xs]):-
   xmember(X, Xs).
```

List Member Log

```
?- ['xmember.pl'].
% xmember.pl compiled 0.00 sec, 584 bytes
Yes
?- xmember(2, [1, 2, 3]).
Yes
?- xmember(4, [1, 2, 3]).
No
?- xmember(X, [1, 2, 3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

List Member Log Continued

```
?- xmember(f(X, a), [f(1, Y), g(2), f(3, Z)]).
X = 1
Y = a
Z = _G162 ;
X = 3
Y = _G151
Z = a ;
No
?- xmember(a, Z).
Z = [a|_G198] ;
Z = [_G197, a|_G201] ;
Z = [_G197, _G200, a|_G204] ;
Z = [_G197, _G200, _G203, a|_G207]
Yes
```

List Member Log Continued

```
?- xmember(X, Y).
X = _G147
Y = [_G147|_G213] ;
X = _G147
Y = [_G212, _G147|_G216] ;
X = _G147
Y = [_G212, _G215, _G147|_G219]
Yes
?-
```

Permutations

Permutations:

```
perms([], []).
perms([X|Xs], Zs):-
  perms(Xs, Ys),
  insert(X, Ys, Zs).

insert(X, Ys, [X|Ys]).
insert(X, [Y|Ys], [Y|Zs]):-
  insert(X, Ys, Zs).
```

Permutations Log

```
?- perms([1, 2, 3], Z).
Z = [1, 2, 3] ;
Z = [2, 1, 3] ;
Z = [2, 3, 1] ;
Z = [1, 3, 2] ;
Z = [3, 1, 2] ;
Z = [3, 2, 1] ;
No
```

Permutations Log Continued

```
?- perms(X, [1, 2]).

X = [1, 2] ;

X = [2, 1] ;

Action (h for help) ? a
abort
% Execution Aborted
?-
```

Quick Sort

Quick sort program:

```
qsort([], []).
qsort([X|Xs], Zs):-
  partition(Xs, X, LEs, GTs),
  qsort(LEs, SortedLEs),
  qsort(GTs, SortedGTs),
  append(SortedLEs, [X|SortedGTs], Zs).

partition([], _, [], []).
partition([X|Xs], E, [X|LEs], GTs):-
  X @=< E,
  partition(Xs, E, LEs, GTs).
partition([X|Xs], E, LEs, [X|GTs]):-
  X @> E,
  partition(Xs, E, LEs, GTs).
```

Quick Sort Log

```
?- qsort([c, b, 1, 2, 'X'], Z).
Z = [1, 2, 'X', b, c] ;
No
?- qsort([f(x), 2, Y, 1, f(g)], Z).
Y = _G155
Z = [_G155, 1, 2, f(g), f(x)] ;
No
?- qsort(X, [1, 2, 3]).
X = [1, 2, 3] ;
Action (h for help) ? a
abort
% Execution Aborted
```

Arithmetic

- `is/2` allows evaluating arithmetic expressions. *N* `is` *E* evaluates arithmetic expression *E* and unifies result with *N* (which is usually a unbound variable at the time of call). Note that *E* cannot contain any variables unbound at the time of call.
- More logical predicates: `plus(`*X*`,` *Y*`,` *Z*`)` can be used with any 2 arguments instantiated.

Arithmetic Log

```
?- N is 1 + 2.
N = 3 ;
No
?- N is 2*3 + 5 mod 2.
N = 7 ;
No
```

Arithmetic Log Continued

```
?- plus(1, 2, Z).
Z = 3 ;
No
?- plus(X, 5, 9).
X = 4 ;
No
?- succ(X, 5).
X = 4 ;
No
?- succ(4, Z).
Z = 5 ;
No
```

Arithmetic Log Continued

```
?- 3 is 1 + M.
ERROR: Arguments are not sufficiently instantiated
?- succ(Z, -1).
ERROR: succ/2: Domain error: \verb@not_less_than_ze
       expected, found @-1'
?- plus(X, Y, 4).
ERROR: succ/2: Arguments are not sufficiently
       instantiated
?-
```

List Length

xlength.pl:

```
xlength([], 0).
xlength([_|Xs], L):-
  xlength(Xs, LXs),
  L is LXs + 1.
```

List Length Log

```
?- xlength([], Z).
Z = 0 ;
No
?- xlength([a, b], Z).
Z = 2 ;
No
?- xlength([a, b], 1).
No
```

List Length Log Continued

```
?- xlength(Z, 2).
Z = [_G203, _G206] ;
Action (h for help) ? a
abort
% Execution Aborted
?- xlength([_, _, _|_], 2).
Action (h for help) ? a
abort
% Execution Aborted
```

Peano Arithmetic

peano.pl:

```
s_number(0).
s_number(s(X)):-
  s_number(X).

add(0, X, X):-
  s_number(X).
add(s(X), Y, s(Z)):-
  add(X, Y, Z).

mult(0, X, 0):-
  s_number(X).
mult(s(X), Y, Z):-
  mult(X, Y, XY),
  add(Y, XY, Z).
```

Peano Arithmetic Log

```
?- add(s(0), s(s(0)), Z).
Z = s(s(s(0))) ;
No
?- add(X, Y, s(s(s(s(0))))).
X = 0
Y = s(s(s(s(0)))) ;
X = s(0)
Y = s(s(s(0))) ;
X = s(s(0))
Y = s(s(0)) ;
X = s(s(s(0)))
Y = s(0) ;
X = s(s(s(s(0))))
Y = 0 ;
No
```

Peano Arithmetic Log

```
?- mult(s(s(0)), s(s(s(0))), Z).
Z = s(s(s(s(s(s(0)))))) ;
No
?- mult(X, Y, s(s(s(s(s(0)))))).
X = s(0)
Y = s(s(s(s(s(0))))) ;
ERROR: Out of global stack
   Exception: (8) mult(_G69, _G27, _L169) ? a
% Execution Aborted
?-
```

Negation-By-Failure

- Negation of a goal is denoted using \+. Not a logical not.
- \+ *G* succeeds iff *G* fails; referred to as *negation-by-failure*.
- When \+ *G* succeeds, no variables in *G* will be bound.
- To avoid surprising results, \+ *G* should be used only when *G* is "ground" (does not contain any variables).
- Can have logical behaviour under the *closed-world assumption*: i.e., all relevant aspects of the world are represented.
- When the system fails to prove something, we assume that the something is false; valid only under the closed world assumption.

Negation Example

negation.pl:

```
mother(joe, jane).
father(joe, frank).


non_parent(X, Y):-
   \+ mother(X, Y),
   \+ father(X, Y).

student(joe).
```

Negation Log

```
?- non_parent(joe, harry).
Yes
?- non_parent(joe, frank).
No
?- non_parent(jill, joe).
Yes
?- non_parent(X, Y).
No
?- student(Y), non_parent(X, Y).
Y = joe
X = _G149 ;
No
?- non_parent(X, Y), student(Y).
No
?-
```

N Queens

Place N queens on a $N \times N$ chess-board such that no queen attacks the other.

- Queens in the same row attack each other; hence assume each queen in a separate row. Specifically, assume solution as a N-element vector `Zs` where `Zs[i]` gives column number of queen in row `i`.
- Queens in the same column attack each other: ensure solution vector contains distinct column number.
- Iterate solution vector to ensure queens do not attack each other via main or auxiliary diagonals.

N Queens Program

In n-queens.pl:

```
n_queens(N, Zs):-
  length(Zs, N),
  place_queens(0, N, Zs).

place_queens(N, N, _Zs).
place_queens(RowIndex, N, Zs):-
  RowIndex < N,
  N1 is N - 1,
  between(0, N1, ColIndex),
  \+attack_queen(0, coord(RowIndex, ColIndex), Zs),
  nth0(RowIndex, Zs, ColIndex),
  RowIndex1 is RowIndex + 1,
  place_queens(RowIndex1, N, Zs).
```

N Queens Program Continued

```
attack_queen(I, coord(RowIndex, ColIndex), Zs):-
  I < RowIndex,
  nth0(I, Zs, IColIndex),
  attack_queen(coord(I, IColIndex),
               coord(RowIndex, ColIndex)).
attack_queen(I, coord(RowIndex, ColIndex), Zs):-
  I < RowIndex,
  I1 is I + 1,
  attack_queen(I1, coord(RowIndex, ColIndex), Zs).
```

N Queens Program Continued

```
%queens attack on same col
attack_queen(coord(_, ColIndex),
             coord(_, ColIndex)).

%queens attack on major diagonal
attack_queen(coord(RowIndex1, ColIndex1),
             coord(RowIndex2, ColIndex2)):-
  ColIndex1 \== ColIndex2,
  RowDiff is RowIndex2 - RowIndex1,
  ColIndex2 is ColIndex1 + RowDiff.

%queens attack on minor diagonal
attack_queen(coord(RowIndex1, ColIndex1),
             coord(RowIndex2, ColIndex2)):-
  ColIndex1 \== ColIndex2,
  RowDiff is RowIndex2 - RowIndex1,
  ColIndex2 is ColIndex1 - RowDiff.
```

N Queens Log

```
?- n_queens(0, Zs).
Zs = [] ;
false.
?- n_queens(1, Zs).
Zs = [0] ;
false.
?- n_queens(2, Zs).
false.
?- n_queens(3, Zs).
false.
?- n_queens(4, Zs).
Zs = [1, 3, 0, 2] ;
Zs = [2, 0, 3, 1] ;
false.
?-
```

River Crossing Puzzle

A farmer who is taking a cabbage, goat and wolf to the market needs to cross a river in a boat which can hold only one item in addition to the farmer. If left unattended, the goat will eat the cabbage and the wolf will eat the goat. How can the farmer get all items across the river?

- Farmer may need to cross river alone or take item back-and-forth across the river. How to avoid an infinite loop?
- Key is to limit the number of trips across the river and increase number of trips across river to find (more) solutions.
- Basically uses iterative depth-first-search.

River Crossing Program

In river-crossing.pl:

```
river_cross:-
  init_state(Init),
  length(Moves, _),   %iterative depth-first search
                      %gen Moves of increasing lengt
  river_cross(Init, Moves),
  out_river_cross_moves(Moves). %output, unimportan

river_cross(State, []):-
  final_state(State).
river_cross(State, [State-Move-StateX|Moves]):-
  \+ final_state(State),
  move(State, Move, StateX),
  \+ forbidden_state(StateX),
  river_cross(StateX, Moves).
```

River Crossing Program Continued

```
%Initialize problem state:
init_state(
   state([f, c, g, w], %farmer, cabbage, goat, wolf
         []             %no one on other bank
        )).

final_state(state([], Bank2)):-
   member(f, Bank2),
   member(c, Bank2),
   member(g, Bank2),
   member(w, Bank2).
```

River Crossing Program Continued

```
forbidden_state(state(Bank1, _Bank2)):-
  forbidden_cohorts(Bank1).
forbidden_state(state(_Bank1, Bank2)):-
  forbidden_cohorts(Bank2).

forbidden_cohorts(Cohorts):-
  \+member(f, Cohorts),
  member(c, Cohorts),
  member(g, Cohorts).
forbidden_cohorts(Cohorts):-
  \+member(f, Cohorts),
  member(g, Cohorts),
  member(w, Cohorts).
```

River Crossing Program Continued

```
%farmer can row herself from Bank1 to Bank2
move(state(Bank1, Bank2), forward(f),
     state(Bank1Z, [f|Bank2])):-
  select(f, Bank1, Bank1Z).

%farmer can row herself from Bank2 to Bank1
move(state(Bank1, Bank2), backward(f),
     state([f|Bank1], Bank2Z)):-
  select(f, Bank2, Bank2Z).
```

River Crossing Program Continued

```
%farmer can row herself + some X from Bank1 to Bank
move(state(Bank1, Bank2), forward(f, X),
     state(Bank1Z, [f, X|Bank2])):-
  select(f, Bank1, Bank1X),
  select(X, Bank1X, Bank1Z).

%farmer can row herself + some X from Bank2 to Bank
move(state(Bank1, Bank2), backward(f, X),
     state([f, X|Bank1], Bank2Z)):-
  select(f, Bank2, Bank2X),
  select(X, Bank2X, Bank2Z).
```

River Crossing Program Continued

```
out_river_cross_moves([]).
out_river_cross_moves([State-Move-StateX|Moves]):-
  format("~w~20+ --~w~16+--> ~w~n",
         [State, Move, StateX]),
  out_river_cross_moves(Moves).
```

River Crossing Log

```
?- river_cross.
state([f,c,g,w],[])   --forward(f,g) --> state([c,w]
state([c,w],[f,g])    --backward(f)  --> state([f,c,
state([f,c,w],[g])    --forward(f,c) --> state([w],[
state([w],[f,c,g])    --backward(f,g)--> state([f,g,
state([f,g,w],[c])    --forward(f,w) --> state([g],[
state([g],[f,w,c])    --backward(f)  --> state([f,g]
state([f,g],[w,c])    --forward(f,g) --> state([],[f
true ;
state([f,c,g,w],[])   --forward(f,g) --> state([c,w]
state([c,w],[f,g])    --backward(f)  --> state([f,c,
state([f,c,w],[g])    --forward(f,w) --> state([c],[
state([c],[f,w,g])    --backward(f,g)--> state([f,g,
state([f,g,c],[w])    --forward(f,c) --> state([g],[
state([g],[f,c,w])    --backward(f)  --> state([f,g]
state([f,g],[c,w])    --forward(f,g) --> state([],[f
true
```

The Cut

- A `cut` denoted using `!` is logically equivalent to `true`.
- A cut is used for *control*, to prevent backtracking. Specifically, once a cut is executed, control cannot backtrack over it.
- Once a cut is executed within a Prolog rule, it denotes commitment to choosing that rule as well as all decisions made in the body of the rule before the cut.
- Should avoid using `!` for *logical* programs, but necessary sometimes for efficiency or to trim unneeded solutions.

Cut Example

```
f(a). f(b). f(c).
g(1). g(2). g(3).
h(x). h(y). h(z).

p1(X):- f(X).
p1(X):- g(X).
p1(X):- h(X).
```

Cut Example Continued

```
p2(X):- f(X).
p2(X):- g(X), !.
p2(X):- h(X).

p3(X):- f(X).
p3(X):- !, g(X).
p3(X):- h(X).
```

Cut Log

```
?- p1(X).
X = a ;
X = b ;
X = c ;
X = 1 ;
X = 2 ;
X = 3 ;
X = x ;
X = y ;
X = z ;
No
```

Cut Log Continued

```
?- p2(X).
X = a ;
X = b ;
X = c ;
X = 1 ;
No
```

Cut Log Continued

```
?- p3(X).
X = a ;
X = b ;
X = c ;
X = 1 ;
X = 2 ;
X = 3 ;
No
```

Cut Log Continued

```
?- p1(x).
Yes
?- p2(x).
Yes
?- p3(x).
No
?-
```

Length Revisited

Previous `xlength/2` went into infinite loop because it kept generating longer lists. Fix with `zlength/2` below.
zlength.pl:

```
zlength([], 0).
zlength([_|Xs], L):-
  zlength(Xs, LXs),
  succ(LXs, L),
  !.
```

Length Revisited Log

```
?- zlength([1, 2], Z).
Z = 2 ;
No
?- xlength(X, 2).
X = [_G203, _G206] ;
Action (h for help) ? a
abort
% Execution Aborted
?- zlength(X, 2).
X = [_G206, _G209] ;
No
?-
```

Taming Cut

- A *green cut* is purely a efficiency enhancing device to increase the efficiency of the program. Example: the use of cut in `zlength/2`, prevented an infinite loop, without loosing any solutions.
- A *red cut* changes the semantics of the program. Example: the program with `p1/1`, `p2/1`, `p3/1` which gave different answers depending on the placement of the cut.

Cut Examples

```
drive(X, Z):- red(X), !, ...
drive(X, Z):- green(X), ...
```

Assuming that `red(X)` and `green(X)` are mutually exclusive, then above cut is a green cut. However, using the fact that if `red(X)` is true, then `green(X)` must be false, the programmer then transforms the program:

```
drive(X, Z):- red(X), !, ...
drive(X, Z):-  ...  %omit green(X) test.
```

the cut becomes red.

Or

`A ; B` means try to solve `A`. If that fails try to solve `B`.

```
(A ; B):- A.
(A ; B):- B.
```

If-Then-Else

- Disciplined use of a cut.
- `A -> B ; C` means if `A` then `B` else `C`.
- Equivalent to:

```
(A -> B ; C):- A, !, B.
(A -> B ; C):- C.
```

- `A -> B` equivalent to if `A` then `B` else `fail`.

If-Then-Else Example

```
merge([], X, X):- !.
merge(X, [], X):- !.
merge([X|Xs], [Y|Ys], Zs):-
   X @< Y ->
      Zs = [X|Z1], merge(Xs, [Y|Ys], Zs1)
   ; Zs = [Y|Z1], merge([X|Xs], Ys, Zs1).
```

Merge Sort

xmsort.pl:

```
xmsort([], []):- !.
xmsort([A], [A]):- !.
xmsort([A|As], Sorted):-
  xsplit([A|As], Odds, Evens),
  xmsort(Odds, SortedOdds),
  xmsort(Evens, SortedEvens),
  xmerge(SortedOdds, SortedEvens, Sorted).

xsplit([], [], []).
xsplit([A|As], [A|Odds], Evens):-
  xsplit(As, Evens, Odds).
```

Merge Sort Continued

```
xmerge([], Xs, Xs):- !.
xmerge(Xs, [], Xs):- !.
xmerge([X|Xs], [Y|Ys], Zs):-
   X @< Y ->
      Zs = [X|Zs1], xmerge(Xs, [Y|Ys], Zs1)
   ; Zs = [Y|Zs1], xmerge([X|Xs], Ys, Zs1).
```

Merge Sort Log

```
?- xsplit([1, 2, 3, 4], X, Y).
X = [1, 3]
Y = [2, 4] ;
No
?- xsplit([1, 2, 3, 4, 5], X, Y).
X = [1, 3, 5]
Y = [2, 4] ;
No
?- xmerge([], [2], Z).
Z = [2] ;
No
```

Merge Sort Log Continued

```
?- xmerge([1, 3, 5], [2, 4], Z).
Z = [1, 2, 3, 4, 5] ;
No
?- xmsort([7, 4, 4, 2, 5, 2, 3], Z).
Z = [2, 2, 3, 4, 4, 5, 7] ;
No
?-
```

Meta Predicates

Various predicates which allow manipulating program facts, rules and relations.

- `clause(`*Head*`,` *Body*`)` : Finds first rule or fact in database matching *Head* and *Body*. For facts, *Body* matches `true`.
- `call(`*G*`)` calls goal *G* which allows building a goal dynamically.

A Prolog Meta-Interpreter

An interpreter for Prolog in Prolog: provides a framework for changing operation (for example, replace DFS with iterative DFS).
In meta.pl:

```
solve(true):-
  !.
solve((Goal, Goals)):-
  !,
  solve(Goal),
  solve(Goals).
solve(Goal):-
  clause(Goal, Body),
  solve(Body).
```

Meta-Interpreter Log

```
?- solve(parent(bill, X)).
X = marge ;
X = harry ;
No
?- solve(ancestor(jill, X)).
X = mary ;
X = frank ;
No
?-
```

Prolog Programming Heuristics

- Approach program declaratively; i.e. concentrate on writing true facts and rules about the domain.
- A Prolog procedure which is collection of facts/rules for a particular head predicate should constitute a complete description for the relation specified by the head predicate.
- If a relation is a function, add additional argument for the result of the function.
- If a procedure is about a particular data-structure, use structural induction to ensure that all variants of the data-structure are covered, possibly by using a separate fact/rule for each variant of the data-structure.
- Often (not always) facts/rules for a procedure need to be mutually-exclusive. Ensured using pattern-matching in the head and/or guard predicates at the start of the body.

Prolog Programming Heuristics

- If two variables must match (be *equal*), simply use the same variable name for both.
- If a statement is false, simply omit writing it (for example, by simply omitting the case for `[]`, `member/2` documents the fact that no element is a member of an empty list). Alternately, you can document the fact that a statement is false by writing something like

```
member(_, []):- false.
```

Prolog Programming Heuristics Continued

Declaratively, order does not matter, but operationally it does matter.

- Write procedure with certain instantiation patterns in mind.
- Order facts/rules within a procedure to maximize the number of instantiation patterns for which program terminates. Usually, facts/rules covering base cases before recursively-defined rules.
- Order the predicates within a body to ensure early failure. Hence guards to ensure that a rule is applicable should be at the start of the body.
- Add cuts, if-then-else only for efficiency, to trim redundant solutions or prevent infinite loops.
- As far as possible, avoid red cuts.

References

Textbook: Ch. 12.
Paul Brna, *Prolog Programming: A First Course*.
SWI-Prolog.
W. F. Clocksin and C. S. Mellish, "Programming in Prolog", 3rd Edition, Springer-Verlag, 1987.