Advice for Writing Prolog

To minimize bugs, especially with cut and negation:

- Use cut or negation as necessary to avoid wrong answers.
- Always use ";" when testing to check all possible answers.
- Use cut to avoid duplicate answers.
- Use cut where possible for efficiency.
- Use "_" where possible for efficiency.
- Follow the safety guidelines for negation.
- Test with variables in every combination of positions.
- Use a precondition to state where variables are disallowed.

Prolog Review

- Logic Programming
- Prolog vs. Scheme (relational vs. functional)
- Logic Programming vs. Prolog (nondeterministic vs. deterministic, e tc.)
- Prolog Syntax (Horn Clauses (w/ variables), Facts, translating from english to Prolog)
- Writing Recursive Predicates (e.g., family relations)
- Lists (internal representation (dot predicate), head/tail)
- Recursive Predicates for List Manipulation (including accumulators)
- Other Structures (functions, e.g., resistor, parse tree, tree, etc.)
- How Prolog Works
  - Unification
  - Goal-Directed Reasoning
  - Rule Ordering
  - Backtracking DFS

- Improving Efficiency
  - Anonymous Variables
  - Accumulators
  - CUT
- Negation as Failure (NAF) (safety conditions, etc.)
- Arithmetics
- Cut (!)
- Beyond the Course:
  - univ, call, functor, arg, assert, retract
  - Nondeterministic Programs

...Beyond what we're "Officially" Covering, but interesting nonetheless!

univ

The standard built-in predicate called 'univ' (=..) translates a predicate and its arguments into a list whose first element is the predicate name and whose remaining elements are the arguments. It works in reverse as well.

For example,

```
?- pred(arg1,arg2) =..  X.
X = [pred, arg1, arg2]

?- pred =..  X.
X = [pred]

?- X =..  [pred,arg1,arg1].
X = pred(arg1, arg2)

?- X =..  [pred].
X = pred
```


Example using univ

Define polygons figures as follows:

```
square(Side)
triangle(Side1,Side2,Side3)
circle(R)
...
```

We'd like to define a predicate that enlarges each of these figures.

```
enlarge(Fig,Factor,Fig1).
```

Here's one way:

```
enlarge(square(A),F,square(A1) :-
  A1 is F*A.
enlarge(circle(R),F,circle(R1) :-
  R1 is F*R1.
...
```

Using **univ**, we can do it much more elegantly:

```
enlarge(Fig,F,Fig1) :-
  Fig=..[Type|Parameters],
  multiplylist(Parameters,F,Parameters1),
  Fig1=..[Type|Parameters1].

multiplylist([],_,[]).

multiplylist([X|L],F,[X1|L1]) :-
  X1 is F*X, multiplylist(L,F,L1).
```


call, functor, arg

call allows you to call a predicate. E.g.,

```
Goal=..[Functor | Arglist].
call(Goal).
```

Alternatively, you can do this with **functor** and **arg**.

```
functor(Term,F,N)
```

functor is true if F is the principal functor of Tern and N is the arity of F.

```
arg(N,Term,A)
```

arg is true if A is the Nth argument in Term, assuming that arguemnts are numbered from left to right starting with 1.

E.g.,

```
?- functor(t(f(X),X,t),Fun,Arity).
   Fun=t
   Arity=3

?- arg(2,f(X,t(a),t(b)),Y).
   Y=t(a)

?- functor(D,examdate,3),
   arg(1,D,22),
   arg(2,D,april),
   arg(3,D,2004).

   D=examdate(22,april,2004)
```


assert/retract

Here is an example illustrating how clauses may be added and deleted from the Prolog data base. The example shows how to simulate an assignment statement by using **assert** and **retract** to modify the association between a variable and a value.

```
:- dynamic x/1 .

x(0).          % provide an initial value

assign(X,V) :- Old =..[X,_], retract(Old),
               New =..[X,V], assert(New).
```

Here is an example using the assign predicate.

```
?- x(N).

N = 0
Yes
?- assign(x,5).
Yes
?- x(N).

N = 5
```

Code with structures

Suppose we represent ordinary binary trees as follows:

- the atom "empty" represents an empty binary tree
- the structure node(K,L,R) represents a tree with integer value K at the root, left subtree L, and right subtree R.

Write these predicates:

- member
- member for a binary *search* tree
- insert for a binary search tree
- delete for a binary search tree

Program as Data

In Prolog, a predicate is a structure. Example:

```
second(hello, X)
```

There is no structural difference between a query and data. (But we can execute a structure that represents a query.)

This allows us to build up a query, or tear one apart and modify it — and then execute the result.

Exercise: Write this predicate:

```
; compare(Table, Row, Column, Comparison, Value)
; succeeds iff the value of Table in position (Row, Col)
; compares to Value according to the given Comparison.
; Precondition: Comparison is a predicate of two arguments.
```

Example:

```
| ?- compare([[1,2,3],[4,5,6],[7,8,9]], 1, 3, <, 27).
yes
| ?- compare([[1,2,3],[4,5,6],[7,8,9]], 1, 3, <, 1).
no
```


Univ and Call

Two built-in Prolog predicates:

Univ: for building queries

The goal

```
X =.. L
```

succeeds iff X is a structure whose functor is the first element of L, and whose arguments are the remaining elements of L.

Call: for executing queries

The goal

```
call(X)
```

succeeds iff X succeeds.

This seems pointless. Why not just write X instead of call(X)?

Let's Write More Predicates

```
% allHave( List, Key ) succeeds iff List is a List,
% and every element of List is itself a list that
% contains Key.


% allListsHave( List, Key ) succeeds iff List is a List,
% and every element of List that is itself a list,
% contains Key.
```

Nondeterministic Programming

(Sterling and Shapiro - the Art of Prolog)

Nondeterminism is powerful for defining and implementing algorithms.

Intuitively, a nondeterministic machine can choose its next operation correctly when faced with several alternatives.

Nondeterminism can be simulated/approximated by Prolog's sequential search and backtracking. Nondeterminism cannot *truly* be achieved.

Examples of nondeterministic programs (mostly for NP-complete problems):

- generate-and-test
- N-queens
- Map colouring
- AI planning
- Towers of Hanoi
- etc.

Towers of Hanoi

Setup: 3 pegs ("left", "centre", "right"). In the initial state one peg (let's say the "left" peg) has N rings on it, stacked from largest to smallest.

Task: Move N disks from the left peg to the right peg using the centre peg as an auxiliary holding peg. At no time can a larger disk be placed upon a smaller disk.

Solution:

```
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl.
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```


Towers of Hanoi (cont.)

```
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl.
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```

Execution for N=3:

```
?-  move(3,left,right,centre).
Move top disk from left to right
Move top disk from left to centre
Move top disk from right to centre
Move top disk from left to right
Move top disk from centre to left
Move top disk from centre to right
Move top disk from left to right

Yes
```


Towers of Hanoi (cont.)

```
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl.
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```