# Lesson #3: Principles of Prolog

*Principle: A basic idea that explains how something works.*

## Example: Prefiguring Prolog

### Task

Show that P is a logical consequence of:

1. $( Q \wedge T ) \rightarrow P$
2. $R \rightarrow Q$
3. R
4. T

### Work

1. Translate the given WFFs to clausal from
   (a) $\sim Q \vee \sim T \vee P$ (*switcheroo and DeMorgan*)
   (b) $\sim R \vee Q$ (*switcheroo*)
   (c) R
   (d) T
2. Add the negation of the goal to the set of clauses
   (a) $\sim Q \vee \sim T \vee P$
   (b) $\sim R \vee Q$
   (c) R
   (d) T
   (e) $\sim P$
3. Refute!
   (a) $\sim Q \vee \sim T \vee P$
   (b) $\sim R \vee Q$
   (c) $\sim T \vee P \vee \sim R$
   (d) R
   (e) $\sim T \vee P$
   (f) T

(g) P

(h) $\sim$ P

(i) $\square$

## Prelude to Prolog

What we did in the previous items reflects in an essential way how you engage in Prolog programming. You provide Prolog with an acceptable set of statements (restricted WFFs) and a goal (restricted WFF), and Prolog does the rest! Breaking this down a bit:

1. You provide Prolog with a knowledge base of statements (restricted WFFs) in the form of "facts" and "rules". Prolog will convert these to restricted clauses for you – **Horn clauses**, in paricular, which are simply clauses with at most one unnegated literal. (The form of Prolog statements assures that all of the converted clauses will be Horn clauses.)
2. You provide prolog with a goal. Prolog will negate it for you and add it to the set of clauses.
3. Prolog will perform the refutation – or at least try its best.

Prolog is essentially a "Horn clause problem solver", where a Horn clause is , as has been mentioned, a clause with at most one unnegated literal.

The power of Horn clause problem solving (and Prolog) comes from the scaling up of propositional calculus to predicate calculus – which amounts to adding variables (and a host of ideas that come with them) to the propositional calculus. We will consider this very briefly, ever so briefly, in just a bit.

## Prolog Demo: Previous exercise/example in Prolog

### Code

```
p :- q,t.
q :- r.
r.
t.
```

### Demo

```
bash-3.2$ swipl
...
?- consult('example1.pro').
% example1.pro compiled 0.00 sec, 5 clauses
true.

?- p.
true.
```

```
?-
```

## Notes

1. Note that the symbol :- is read “if”, that the comma to the right of the :- symbol is read “and”, and that the following mean the same thing:
   (a) Prolog: `p :- q, t.`
   (b) Standard logic: $( Q \wedge T ) \rightarrow P$
2. Note also that this form of WFF can be converted to a horn clause (with switcheroo and DeMorgan), a clause with just one unnegated literal: $( \sim Q \vee \sim T \vee P )$

## Prolog Demo: Fruit

### Code

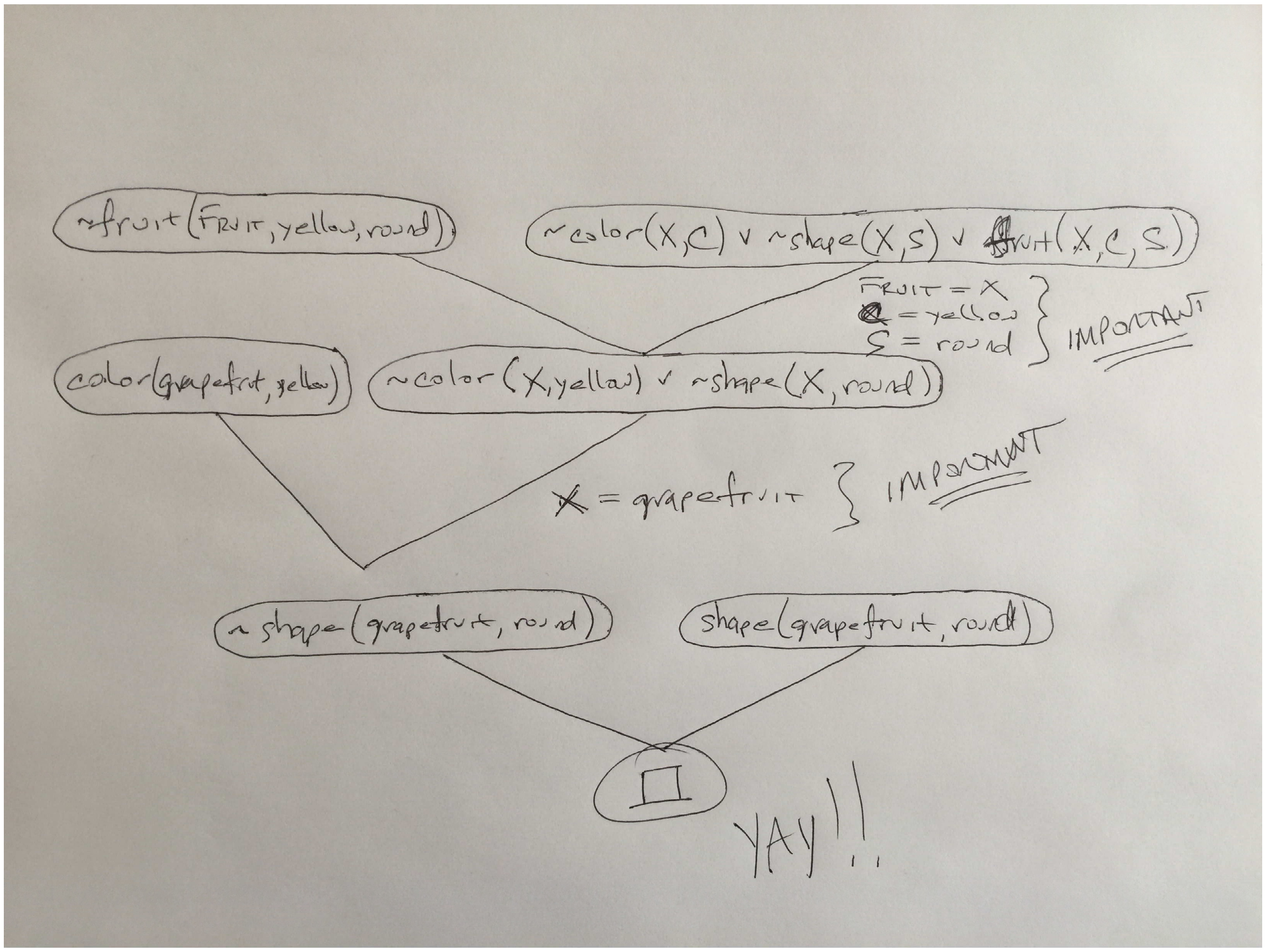
```
% KB of fruit
% 6 facts
  color(apple,red).
  color(banana,yellow).
  color(grapefruit,yellow).
  shape(banana,oblong).
  shape(grapefruit,round).
  shape(apple,round).
% 1 rule
  fruit(X,C,S) :- color(X,C), shape(X,S).
```

### Demo

```
 bash-3.2$ swipl
...
?- consult('fruit.pro').
% fruit.pro compiled 0.00 sec, 8 clauses
true.

?- color(Fruit,red).
Fruit = apple.

?- color(apple,Color).
Color = red.

?- fruit(Fruit,yellow,round).
```

```
Fruit = grapefruit.

?- fruit(Fruit,yellow,oblong).
Fruit = banana

?- fruit(banana,Color,Shape).
Color = yellow,
Shape = oblong.

?- fruit(Fruit,red,oblong).
false.

?-
```

## Fruit Tree


~fruit(FRUIT,yellow,round)
~color(X,C) v ~shape(X,S) v fruit(X,C,S)
FRUIT = X
C = yellow
S = round
IMPORTANT
color(grapefruit,yellow)
~color(X,yellow) v ~shape(X,round)
X = grapefruit
IMPORTANT
~shape(grapefruit,round)
shape(grapefruit,round)
□
YAY!!