Programming Paradigms

Logic Programming using Prolog

Department of Computer Science and Information Systems
Birkbeck, University of London

October 8, 2014

Logic Programming

- Programming languages for logic programming are very different to those encountered so far — they are *declarative* languages
- You present the facts and inference rules and the program will do the reasoning
- In a declarative language
  - the programmer specifies a goal to be achieved
  - the system then *works out* how to achieve it
- In imperative and object-oriented languages, the programmer has to do both

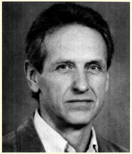
Prolog

One of the most well-known logic programming languages is Prolog

- Stands for **Pro**grammation en **Log**ique (Programming in Logic)
- Developed by Alain Colmerauer and colleagues in the early 1970s
- University of Edinburgh a major player (Clocksin and Mellish) together with Imperial College

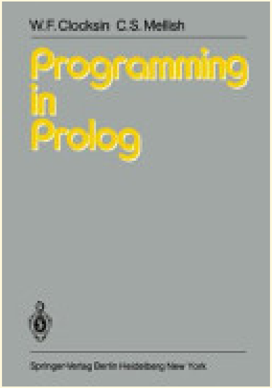
Relational databases owe something to Prolog

Alain Colmerauer



Robert Kowalski

The Book



W.F. Clocksin C.S. Mellish
Programming in Prolog
Springer-Verlag Berlin Heidelberg New York

Mathematical Foundation

Just a brief explanation how Prolog fits into the framework of mathematical logic

- *First-order logic* is a powerful mathematical tool for formalizing descriptions
  - It is also sometimes called *predicate logic*
- Unfortunately, first-order logic is not decidable
- Prolog is based on a decidable subset of first-order logic called *Horn clauses*
- It is still Turing-complete, though

First Program

Let's have a look at a very simple program: Hello World!

```
?- write('Hello World!'), nl.
```

with output[1]:

```
Hello World!

yes
```

Although this works, it's an atypical example of a Prolog program

[1](may say true instead of yes dependent upon the version of Prolog used)

Language Basics

Prolog has two aspects:

- One to express the data
- Another to query the data

Data is represented in the form of facts and logical rules

Facts: a fact is a basic assertion about some world
e.g., Babe is a pig; pigs like mud

Rules: a rule is an inference about facts in that world
e.g., An animal likes mud if it is a pig

Query: a query is a question about that world
e.g., Does Babe like mud?

Data

Facts and rules go into a *knowledge base*

- Prolog allows you to express the contents of a knowledge base
- Usually a compiler turns this base into a form efficient for querying

Querying links together facts and rules to tell you something about the world modeled in the knowledge base

Simple Knowledge Base and Queries I

First some remarks about syntax:

- If a word begins with a lower-case character, it's an *atom*
- An atom is a fixed value, similar to a Ruby symbol
- If it begins with an upper-case letter (or an underscore), it's a *variable*

Simple Knowledge Base and Queries II

Let's have a look at a very simple knowledge base

```
likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).
friend(X,Y) :- likes(X,Z),likes(Y,Z),\+(X=Y).
```

The first five statements are facts, the last one is a rule

Facts I

- In the facts on the previous slide, `wallace`, `gromit`, `wendolene`, `toast`, `cheese`, `cake`, and `sheep` are atoms
- The facts can be read as

  "Wallace likes toast"

  "Wallace likes cheese"

  "Gromit likes cheese"

  "Gromit likes cake"

  "Wendolene likes sheep"

Facts II

- The name of a relationship (before the round brackets) is called a *predicate* (e.g., the predicate `likes` has two parameters)
- The order of atoms in a fact is important, e.g., "cheese likes Wallace" is not a fact

We are now ready to ask some questions...

Queries I

- The most basic queries are questions about facts with a *yes/no* answer
- The following queries are quite intuitive where Prolog tries to match a query to known facts

```
?- likes(wallace,sheep).
no
?- likes(gromit,cheese).
yes
```

- Not very exciting — Prolog is just throwing the facts back at us

Let's try something else...

Queries II

Some atom that isn't in the knowledge base...

```
?- likes(fluffles,sausage).
no
```

So `no` actually means that Prolog cannot prove this statement given the current state of the knowledge base

Instantiation I

- We can ask Prolog to find values for variables:

```
?- likes(Who,cheese).
```

- `Who` is an *uninstantiated* variable, i.e., it does not have a value assigned to it
- Prolog searches the knowledge base from the beginning trying to find a matching fact
- The first matching fact found is `likes(wallace,cheese)`, so `Who` is *instantiated* with `wallace`
- At this point Prolog outputs `Who = wallace`, stops, and asks us what to do

Instantiation II

We can then either

(i) stop searching by just hitting the return key, or

(ii) continue searching by entering `;`

If we continue, Prolog

(i) forgets the value `wallace` for the variable `Who`

(ii) and continues at the position it previously stopped

Continuing will output `Who = gromit` and then `no` (when it finds no further solutions)

Goals I

- By submitting a query, we ask Prolog to try to satisfy a *goal*
- We can ask Prolog to satisfy the conjunction of two goals:

```
?- likes(wallace,toast),likes(gromit,toast).
no
```

- We can combine conjunctions with variables to make queries more interesting
- Now that we found out that at least one of them does not like toast...

Goals II

- . . . is there something both of them like?

```
?- likes(wallace,What),likes(gromit,What).
What = cheese ? ;
no
```

- How does Prolog process this query (conceptually)?
- It uses backtracking to try to satisfy the first goal and then the second goal

Backtracking I

```
likes(wallace,What), likes(gromit,What).
```

first goal — second goal

```
likes(wallace,cheese).

likes(wallace,toast).

likes(gromit,cheese).

likes(gromit,cake).

likes(wendolene,sheep).
```

Backtracking II

likes(wallace,What) , likes(gromit,What).

toast toast

likes(wallace,toast).

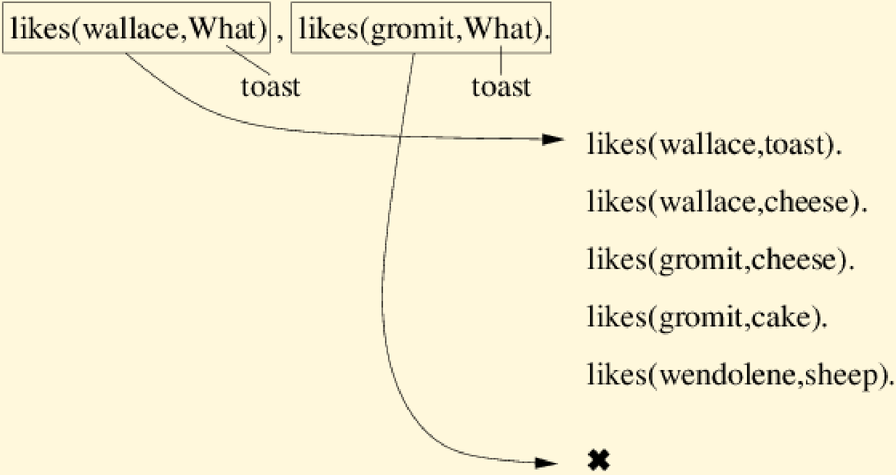
likes(wallace,cheese).

likes(gromit,cheese).

likes(gromit,cake).

likes(wendolene,sheep).

Backtracking III



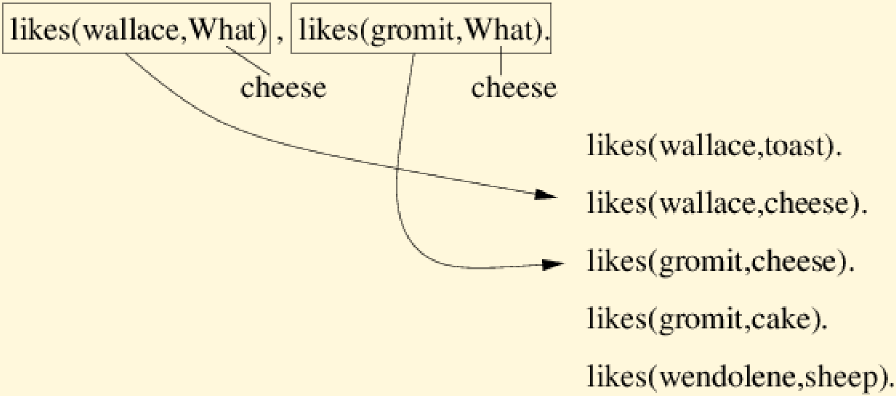
Backtracking IV

```
likes(wallace,What) , likes(gromit,What).
            cheese               cheese
```

```
likes(wallace,toast).
likes(wallace,cheese).
likes(gromit,cheese).
likes(gromit,cake).
likes(wendolene,sheep).
```

Birkbeck UNIVERSITY OF LONDON

Backtracking V



Rules I

- A *rule* is a general statement about objects and their relationships
- A rule in plain English could be:

  X is a sister of Y if:
  X is female and
  X and Y have the same parents.

- Important: a variable stands for the same object wherever it occurs in a rule

Rules II

- Rules in Prolog consist of a *head* and a *body* connected by the symbol `:-` (which is pronounced *if*)
- The predicate `friend` in our knowledge base is defined by a rule
- Predicates can be defined by a combination of facts and rules
- A `clause` of a predicate is a fact or rule defining the predicate

Rules III

- If we want to express that

  "Wallace is a friend of anyone who likes cheese"

  we could formulate it like this:

```
friend(wallace,X) :- likes(X,cheese).
```

- Running the query `friend(wallace,X).` will produce two results: `wallace` and `gromit`
- We can exclude `wallace` by saying that `X` shouldn't be `wallace`

Rules IV

```
friend(wallace,X) :- likes(X,cheese),
                     \+(X=wallace).
```

- `\+` is the negation of a subgoal
- This only lists friends of `wallace` (those persons who like cheese)
- A generalization of the rule would be

  "X and Y are friends, if X and Y like the same Z and X and Y are not the same"

```
friend(X,Y) :- likes(X,Z),likes(Y,Z),\+(X=Y).
```

Rules V

Let's try it out:

```
?- friend(gromit,wallace).
yes
?- friend(wallace,gromit).
yes
?- friend(wallace,wallace).
no
?- friend(wallace,wendolene).
no
```

Rules VI

Now let's ask who is a friend of Wallace:

```
?- friend(wallace,Who).
Who = gromit ? ;
no
```

Or let's find all pairs of friends:

```
?- friend(Who1,Who2).

Who1 = wallace
Who2 = gromit ? ;

Who1 = gromit
Who2 = wallace ? ;

no
```

Another example: from Bratko

...

Example — The Map Colouring Problem I

Just using facts, rules, and variables we can already do some interesting things

Assume we want to colour a map, such that two regions with a common border don't have the same colour

Example — The Map Colouring Problem II

- In order to simplify things, we'll only look at regions 3,4,5, and 6 and use the colours red, green, and blue
- Now all we have to do is describe this to Prolog:

```
border(r,g). border(r,b).
border(g,r). border(g,b).
border(b,r). border(b,g).

colouring(L,TAA,V,FVG) :-
    border(L,TAA),
    border(L,V),
    border(TAA,V),
    border(V,FVG).
```

Example — The Map Colouring Problem III

Querying `colouring(L,TAA,V,FVG).` will now provide all the answers:

```
?- colouring(L,TAA,V,FVG).
FVG = r
L = r
TAA = g
V = b ? ;

FVG = g
L = r
TAA = g
V = b ?
...
```

Where's the Program?

In Prolog you don't have to write a program

- You express the logic of a problem in facts and inferences
- And then let the computer do the work in figuring out a solution

Solving the map colouring problem with a language like Java or Ruby would be much harder to do

Anonymous Variables

- Sometimes we want to use a variable but don't care with which value it is instantiated (We don't want to use the variable anywhere else)
- For example, we want to find out if there is anyone who likes cheese (but we don't need to know who)

```
 ?- likes(_,cheese).
true ?
yes
```

- We use an underscore _ for the anonymous variable
- Several occurrence of _ in the same clause do not need to be given consistent interpretations

Structures I

- If we want to say that Wallace and Wendolene own books, we could formulate the following facts

```
owns(wallace, book).
owns(wendolene, book).
```

- However, this means that Wallace owns the same object that Wendolene owns
- Specifying the title to distinguish may not help:

```
owns(wallace, perfume).
owns(wendolene, russell_the_sheep).
```

- It's not clear we are talking about books here

Structures II

- We can introduce a *structure* for books
- A structure is the closest construct to a data type in Prolog
- A term can be decomposed into its components:

```
owns(wallace,book(perfume,suesskind)).
owns(wendolene,book(russell_the_sheep,scotton)).
```

- Looking at `book(perfume,suesskind)`
  - `book` is the *functor* of the structure
  - `perfume` and `suesskind` are its *components*

Structures III

- Structures can be nested (arbitrarily deep):

```
owns(gromit,book(wuthering_heights,
                 author(emily,bronte))).
```

- We can use structures in querying:
- For example, if we want to know if Gromit owns any books written by one of the Brontë sisters, we would query:

```
?- owns(gromit,book(X,author(Y,bronte))).

X = wuthering_heights
Y = emily
```

Structures IV

- The syntax for structures looks identical to that for facts
- A predicate is actually the functor of a structure
- The arguments of a fact or rule are components of a structure

Equality and Matching I

- Prolog has a number of built-in predicates
- One of them is *equality* written as "="
- Following Prolog syntax, it should be written as `=(X,Y)`
  - While the above works, Prolog also allows you to use an infix notation: `X=Y`
- Prolog attempts to match `X` and `Y`, the goal succeeds if they match

Equality and Matching II

- Integers and atoms are always equal to themselves:

```
?- wallace = wallace.
yes
?- cheese = cake.
no
?- 1066 = 1066.
yes
?- 1206 = 1583.
no
```

- A variable always matches itself:

```
?- X = X.
yes
```

Equality and Matching III

- If we match two different variables, e.g. `X = Y`, we have to distinguish two cases
  1. None or one variable is instantiated
  2. Both are instantiated

Case 1: as soon as one them is instantiated with a value, the other will be instantiated with the same value

```
?- X = Y, likes(X,toast).
X = wallace
Y = wallace ?
yes
```

Birkbeck UNIVERSITY OF LONDON

Equality and Matching IV

Case 2: if both are already instantiated, then it depends on the value they are instantiated with

```
?- likes(X,cheese),likes(Y,cake),X=Y.
X = gromit
Y = gromit ?
yes
?- likes(X,toast),likes(Y,cake),X=Y.
no
```

Comparison and Matching

- Prolog also offers other comparison operators:

```
?- 2 > 3.
no
?- 3 >= 2.
yes
?- 3 =< 2.
no
?-  X \= Y.
no
```

- The last one means that `X` cannot be made equal to `Y`
- You cannot redefine built-in predicates, stating the following as a fact will raise an error

```
2 > 3.
```

Arithmetic I

- Prolog also offers the standard arithmetic operators: +, −, ∗, /, `mod`,
- Just typing in an arithmetic operation will not actually carry it out

```
?- 7 = 3 + 4.
no
```

- Using the `is` operator will evaluate the right-hand side and match it to the left-hand side

```
?- 7 is 3 + 4.
yes
```

Birkbeck UNIVERSITY OF LONDON

Arithmetic II

- Given the following fact base, compute the population density of countries:

```
pop(usa,313).
pop(italy,61).
pop(uk,63).
area(usa,9.826).
area(italy,0.301).
area(uk,0.243).
```

- The following rule computes the density:

```
density(X,Y) :- pop(X,P),area(X,A),Y is P/A.
```

Birkbeck UNIVERSITY OF LONDON

Arithmetic III

- Compute population density of USA:

```
?- density(usa,Y).
Y = 31.854264197028289
yes
```

- Compute all densities:

```
?- density(X,Y).
X = usa
Y = 31.854264197028289 ? ;
X = italy
Y = 202.65780730897012 ? ;
X = uk
Y = 259.25925925925924
yes
```

Lists I

- We have already seen structures as a construct to build more complicated data types
- Another important type supported by Prolog is a list
- The elements of a list are enclosed in square brackets:

```
?- [1,2,3] = [1,2,3].
yes
?- [1,2,3] = [X,Y,Z].
X = 1
Y = 2
Z = 3
yes
```

Lists II

- We can split lists into a *head* and *tail* using the "|" operator:

```
?- [Head|Tail] = [1,2,3].
Head = 1
Tail = [2,3]
yes
?- [Head|Tail] = [].
no
?- [Head|Tail] = [1].
Head = 1
Tail = []
yes
```

Recursion

- Let's assume we want to find out if an element is part of a list
- We have to do this recursively in Prolog
- Recursion in Prolog means that a predicate appears on the left- and the right-hand side of a rule
- For example: an element is *in* a list if it is
  - the head of the list
  - *in* the tail of the list

```
is_in(X,[X|_]).
is_in(X,[_|Y]) :- is_in(X,Y).

?- is_in(d,[a,b,c,d,e,f]).
true
```

Birkbeck UNIVERSITY OF LONDON

Let us take a closer Look I

- You might have noticed that in the book by Bruce Tate, the `friend` rule was written differently:

```
friend(X, Y) :- \+(X=Y),likes(X,Z),likes(Y,Z).
```

- Might not look like a big change, but this has consequences
- For example, if we run the query `friend(wallace,Y).` with the above rule, we get

```
?- friend(wallace,Y).
no
```

- What is going on here?

Let us take a closer Look II

- The position of the predicate `\+(X=Y)` has a big impact
- Prolog tries to satisfy subgoals from left to right
- `\+(X=Y)` fails if `X=Y` can be satisfied
  1. `X` and `Y` start off uninstantiated in the above case
  2. As soon as one of them is instantiated, the other will take on the same value
  3. This makes `X=Y` true, resulting in `\+(X=Y)` being false
  4. Consequently, the first subgoal always fails

Let us take a closer Look III

- If we arrange the predicates in a different order

```
friend(X, Y) :- likes(X,Z),likes(Y,Z),\+(X=Y).
```

- then `X` and `Y` will already be instantiated when reaching the subgoal `\+(X=Y)`
- If `X` and `Y` have a different value at that point, then `\+(X=Y)` will succeed
- It is important to get the order right in which variables are instantiated!

"Cutting" the Number of Solutions I

If you ask Prolog to keep looking for further solutions (by answering with ;) it will go through all possible solutions using backtracking:

```
dance_pairs(X,Y) :- boy(X), girl(Y).
boy(adam).
boy(bert).
...
girl(angela).
girl(betty).
...
?- dance_pairs(X,Y).
X = adam, Y = angela ;
X = adam, Y = betty ;
...
```

"Cutting" the Number of Solutions II

Sometimes we are not interested in exhaustively going through all solutions:

- We only want to know if a solution exists
- We are happy with a certain subset of solutions
- In some recursive cases, there may be an infinite number of solutions

Prolog provides the cut operator to force it not to consider certain choices

The Cut Operator I

- The cut operator is denoted by `!` and can be inserted into a rule as a subgoal
- What does it do? Let's have a look:

```
foo :- a,b.
foo :- c,d,!,e,f.
foo :- g,h.
```

- First of all, `!` always succeeds, i.e., if `c` and `d` are satisfied in the second rule, then Prolog will immediately start matching `e`
- But there's more to it...

The Cut Operator II

- Assuming `c` and `d` are satisfied while checking the second rule, then the choices made for `c` and `d` are "locked in"
  - Prolog may not go back and search for other solutions for `c` and `d`
  - It may still do backtracking for `e` and `f`, though
- In addition to this, if the second rule fails, Prolog may not go beyond this rule to try to satisfy `foo`
  - It will not try out `foo :- g,h.`
- How is the cut operator used in practice?

Birkbeck UNIVERSITY OF LONDON

Confirming choice of a rule I

The first use is to tell Prolog that it has found the right rule to apply

- Assume we want to add up the numbers from 1 to N

```
sum_to(1,1).
sum_to(N,Result) :- TmpN is N-1,
                    sum_to(TmpN,TmpRes),
                    Result is TmpRes + N.
```

- While this works, it may start an infinite recursion:

```
?- sum_to(3,X).
X = 6 ? ;
Fatal Error: local stack overflow
```

Confirming choice of a rule II

- Asking for another solution forces Prolog to search for another solution for `sum_to(1,TmpRes)`, applying the rule `sum_to(N,Result)` to it
- Applying the rule will search for a solution for `sum_to(0,TmpRes)`, which in turn will again apply the rule
- Next attempt at satisfying will be to try to match `sum_to(-1,TmpRes)` and so on

Confirming choice of a rule III

- We want to tell Prolog that once it has matched the fact `sum_to(1,1).` it should not try searching for further solutions
- We can achieve this by rewriting the fact:

```
sum_to(1,1) :- !.
sum_to(N,Result) :- TmpN is N-1,
                    sum_to(TmpN,TmpRes),
                    Result is TmpRes + N.

?- sum_to(3,X).
X = 6
yes
```

Confirming choice of a rule IV

- We could just tell Prolog to stop searching for further solutions in the above example
- However, this may not always be under our control

```
go :- sum_to(1,X), foo(apples).
?- go.
```

- If `foo(apples)` fails, then this will trigger backtracking on `sum_to(1,X)`

"Cut-Fail" Combination I

The second use of the cut operator involves the built-in `fail` predicate that cannot be satisfied:

```
p(X) :- fail.
?- p(X).
no
```

Let us consider an example which tries to figure out the correct tax rate for people

"Cut-Fail" Combination II

- Let us define a predicate for the average tax rate
- However, there is a special tax rate for non-residents, i.e., they never pay the average rate

```
average_tax_rate(X) :- non_resident(X),fail.
average_tax_rate(X) :- ...
```

- This will not work, as a non-resident will fail the first rule and then one of the following rules will be applied
- However, that's exactly what we don't want to happen
- The following will make sure that none of the following rules will be applied

```
average_tax_rate(X) :- non_resident(X),!,fail.
average_tax_rate(X) :- ...
```

Generate and Test I

A common programming pattern in Prolog is "generate and test"

```
foo :- g1, g2,..., gn, t1, t2,..., tm.
```

The sequence of predicates `g1, g2,..., gn` can succeed in many different ways

- They generate lots of different potential solutions

The sequence of predicates `t1, t2,..., tm` tests whether something generated by `g1, g2,..., gn` is actually a solution

- If something is not a solution, this causes `g1, g2,..., gn` to backtrack and generate next candidate

Generate and Test II

Example: We want to define integer division just using addition and multiplication

1. Build a predicate that generates all integers:

```
is_integer(0).
is_integer(X) :- is_integer(Y), X is Y+1.
```

2. Then we check the numbers generated by `is_integer`

```
idiv(X,Y,Result) :- is_integer(Result),
                    Prod1 is Result*Y,
                    Prod2 is (Result+1)*Y,
                    Prod1 =< X, Prod2 > X,
                    !.
```

Generate and Test III

- The first line in `idiv` is the generator, the other lines are implementing the test
- We know that there can only be one possible solution
- After reaching it, we can stop the search, otherwise `is_integer` would keep on producing potential `Results`

Cutting too Deeply I

- The cut operator is a dangerous tool and should be used sparingly
- It can behave in unexpected ways.
- We want to formulate that every person has two parents, except `Adam` and `Eve` who have no parents

```
parent(adam,0) :- !.
parent(eve,0) :- !.
parent(X,2).
?- parent(eve,X).
X = 0
?- parent(john,X).
X = 2
?- parent(eve,2).
yes
```

Cutting too Deeply II

- It is considered good programming style to replace cuts by the use of negation (if possible)

```
parent(adam,0).
parent(eve,0).
parent(X,2) :- \+(X = adam), \+(X = eve).
?- parent(eve,X).
X = 0 ? ;
no
?-  parent(john,X).
X = 2
yes
?- parent(eve,2).
no
```

Cutting too Deeply III

The program computing the sum of the numbers from 1 to N can also be rewritten:

```
sum_to(N,1) :- N =< 1.
sum_to(N,Result) :- N > 1,
                    TmpN is N-1,
                    sum_to(TmpN,TmpRes),
                    Result is TmpRes + N.
```

This also makes it clear which rule to use when

Summary I

Strengths of Prolog

- Prolog is very well suited for application centered around Artificial Intelligence (AI)
  - Natural-language processing
  - AI behavior in games
  - Constraint satisfaction problems, such as time tabling and scheduling
- Prolog (or its descendants) is used in the context of the Semantic Web
  - A variant called Datalog is used in databases
- Also used for simulation and prediction software

Summary II

Weaknesses of Prolog

- Prolog has a steeper learning curve compared to other languages
- Fairly focused niche applications, not really a general-purpose language
- There are scalability issues, the basic matching strategy used by Prolog is computationally expensive
  - Has problems to process large data sets
- It is not as declarative as it seems at first glance
  - If you want to write efficient Prolog programs, you have to know what is going on behind the scenes

Questions...