Logic Programming and Prolog

See **Handout # 11 on Logic History**

Logic has a long-standing history going back to the Greek philosophers. **Aristotle** is regarded as the **father of logic**. He devised systematic methods for analyzing and evaluating arguments. He developed **syllogistic logic** [a kind of logical argument in which one proposition (the conclusion) is inferred from two others (the premises) of a certain form].

For example:

Major premise: All humans are mortal.
Minor premise: Socrates is human.
Conclusion: Socrates is mortal

He also catalogued several fallacies often used in arguments.

A century later another Greek named **Chryssipus** developed **propositional logic** in which the fundamental elements were whole propositions (sentences which were either true or false). He developed procedures to determine the truth or falsity of compound propositions based on the truth or falsity of their components.

After another thousand years **Abelard,** of Heloise and Abelard fame, (1079-1142)**,** distinguished arguments that were valid because of their form **(intentional meaning**) from those that were valid because of their content (**extensional meaning**). Another mediaeval logician **William of Occam** developed, in the fourteen century, **modal logic** which includes concepts such as possibility, necessity, belief, and doubt.

Three hundred years later **Leibnitz** (1646-1716) outlined an ambitious scheme for machine reasoning. He envisioned, but never implemented, a "**Characteristica Universalis,"** a calculus that would cover all thoughts and replace controversy by calculation. His calculus was to cover all knowledge (it presupposed an encyclopedia which would include all what was known so far). His calculus required the knowledge to be stated in systematic form. His goal was to represent knowledge in a form which could be used for mechanized reasoning. Unfortunately, he did not have the tools necessary to implement his ideas. He attempted to formulate a symbolic language that could be used to settle all forms of dispute. This continued with works of nineteen century philosopher and mathematicians such as **Bolzano**, **DeMorgan**, **Boole**, **Venn**, and **Frege**.

In 1854 **Boole** published *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*. His major achievement was the demonstration that the tools of algebra could be modified and applied to logical deduction. **He created the logical equivalent of the arithmetic operators: union, intersection, and not; he also invented the truth table in order to test the truth of the compound propositions** he constructed out of the new logical connectives.

Frege invented the **first complete theory of first-order logic**. **Whitehead** and **Russell** (*Principia Mathematica* - 1910) codified symbolic logic in its present form as a system for reducing mathematics to logic. They are often regarded as the founders of modern formal logic.

Logic is closely associated with computers and programming languages in a number of ways:

- **Boolean Algebra** is used to design circuits
- **Boolean expressions** are used to control the flow of execution of programs
- Logical statements have been used to describe the semantics of programming language constructs (**axiomatic semantics**)
- Logical statements are used as **formal specifications** of the behavior of programs and with axiomatic semantics can be used to prove the correctness of programs in purely mathematical ways

Computers have been used to implement the principles of mathematical logic (automatic deduction systems and automatic theorem provers which turn proofs into computations).

In the 60s and 70s it became apparent that computations can be viewed as kinds of proof and that logical statements (at least in a restricted form) can be viewed as a programming language and executed on a computer, as long as one uses a sufficiently sophisticated interpretive system.

Work by **Robinson, Colmerauer, and Kowalski** led to the programming language Prolog which has remained the main, if not the only, **logic programming language** (those are also called **declarative languages**).

First Order Predicate Calculus

This is the kind of mathematical logic used in logic programming. It is a way of expressing **logical statements** (i.e. those that are either true or false). **Axioms** are logical statements that are assumed to be true and from which other statement can be proven.

Propositions

A **proposition** can be thought of as a logical statement that **may or may not be true**. It consists of objects and the relationship of objects to each other. Formal logic was developed to provide a method for describing propositions, the goal being to allow those formally stated propositions to be checked for validity.

- **Atomic propositions** are the simplest propositions. Examples of atomic propositions are:

      fred is poor
      mary is tall
      fido is smelly

  In Prolog they consist of **compound terms** which are written in a form that looks like a mathematical function. For example:

      poor (fred)
      tall (mary)
      smelly (fred)

A compound term consists of a **functor**. Although it looks like a function name, it is not. It is the name of a relation. It is followed by an ordered list of parameters. A compound term with only one parameter is also called a **1-tuple**, one with two parameters a **2-tuple**, etc.

Propositions have no intrinsic semantics, they mean whatever you want them to mean. For example **“like (bob, steak)”** could mean that Bob likes steak but it could also mean that steak likes Bob, or that Bob and steak are similar in some way.

Examples of logical statements and their translation in predicate calculus:

0 is a natural number	(an axiom)	natural (0)
3 is an integer	(true)	integer (3)
-4 is a natural number	(false)	natural (-4)
Jake is a man	(true)	man (jake)
Mary is a man	(false)	man (mary)
Bob likes steaks	(true)	like (bob, steak)

For all x, if x is a natural number, then so is the successor of x (an axiom)

For all x, natural (x) → natural (successor (x))

As indicated before axioms are statements that are assumed to be true and from which other statement can be proven.

- **compound propositions** have one, two or more atomic propositions connected by logical connectors (¬ negation, ⋀ conjunction, ⋁ disjunction, ⇔ equivalence, and ⇒ implication).

  In first-order predicate calculus the different parts of logical statements are:

  - **Constants**: usually numbers or names (also called **atoms**) (e.g. 0, 3, -4, jake, mary, bob above).
  - **Variables**: stand for as yet unspecified quantities (e.g., x above). It can represent different objects at different times. The binding of a value to a

variable is called an **instantiation**. A variable that has not been assigned a value is called **uninstantiated**.

- **Predicates**: names for functions that are true or false, may take any number of arguments (e.g. natural above is a predicate with one argument).
- **Functions**: functions that are not predicates (e.g. successor above is a function, not a predicate). It consists of a functor (a function symbol that names the proposition) and an ordered list of parameters.
- **Connectives**: operations such as **conjunction/and** (∧), **disjunction/or** (∨), **negation/not** (¬), **implication** (**→/**⊃ or **⇐/**⊂), and **equivalence** (**↔/**≡)

  a → b, or b ⇐ a means that b is true whenever a is and is equivalent to "b **or not** a"

  Note that when a is false, b can either be true or false.

  a ↔ b is equivalent to (a → b) and (b → a)
- **Quantifiers**: these are operations that introduce variables.
  - **For all** (∀) is the **universal quantifier**
  - **There exists** (∃) is the **existential quantifier**

  A variable introduced by the quantifier is said to be **bound** by the quantifier. Variables that are not bound by a quantifier are called **free variables**.
- **Punctuation Symbols**: parentheses, comma, and period (*this one not strictly necessary but typically used in most logic programming systems*.)

In predicate calculus, **arguments to predicates and functions** can only be **terms** (i.e. combinations of variables, constants and functions). *Terms cannot contain predicates, quantifiers or connectives*. For example, in the examples above, 0, 3, -4, x, and successor (x) are terms, ∃ x…, or ∀ y… are not.

In addition, predicate calculus has **inference rules** which are ways to derive (i.e., prove) new statements from a given set of statements. For example:

from (a → b) and (b → c) we can derive (a → c)

Let us look at another example. We will assume the following axioms:

mammal (horse)
mammal (human)

for all x, mammal (x) → legs (x, 4) **and** arms (x, 0)
or legs (x , 2) and arms (x, 2)
arms (horse, 0)
not arms (human, 0)

In this set of statements we have the following:

• constants:	0, 2, 4, horse, human (also called atoms)
• predicates:	mammal, arms, legs
• a variable:	x

- there are no functions

from the five axioms above we can infer (we will see later how)

legs (horse, 4)
legs (human, 2)
arms (human, 2)

these are called **theorems**

This represents the essence of logic programming: a collection of statements that are assumed to be axioms, and from them a desired fact is derived by the application of inference rules in some automated way.

A **logic programming language** is a notational system for writing logical statements together with specified algorithms (hidden in the background "engine") for implementing inference rules. There are two groups of logical statements:

• a set of logical statements taken to be axioms	the **logic program**
• logical statements to be derived	input (**queries** or **goals**)

Example of a query:

Does there exists a **y** such that **y** is the number of legs of a human? (English)
or there exists **y**, legs (human, **y**) (Logic)

logic programming systems are sometimes referred to as **deductive databases** (consisting of a set of statements and a deduction system that can respond to queries).

These databases are significantly different from conventional databases that contain only facts (deductive databases contain not only facts but also implications (e.g. natural (x) → natural(successor(x))).
Pure logic programming systems do not provide any rules about how a particular statement might be derived from a set of statements.

However, in automatic deduction systems, the specific path (sequence of steps) that the system chooses to derive a statement is the **control problem** for a logic programming system and will include an algorithm used for such derivation.

Horn Clauses

Most logic programming systems do not handle all of first-order predicate calculus but restrict themselves to **Horn clauses**.

A Horn Clause is a statement of the form

a_1 and a_2 and a_3 . . . and a_n $\Rightarrow$ b
which is equivalent to $\neg a_1 \lor \neg a_2 \lor \neg a_3 \ldots \lor \neg a_n \lor b$

where $a_1, a_2, \ldots, a_n$ can only be **simple statements** involving no connectives. (hence no “or” and no quantifiers). It says that b is true if all the a_i’s are true

Terminology:

1. **head** of the clause: the right hand side of the implication symbol (i.e. **b**)
2. **body**: the left hand side (i.e. a_1 and a_2 and a_3 . . . and a_n)

If there are no a (i.e. the clause is simply $\Rightarrow$ b), then b is always true, it is an axiom and such clause is sometimes called a **fact**. In addition, the symbol “$\Rightarrow$” is usually omitted.

Horn clauses are of particular interest to automatic deduction systems such as logic programming systems, because they can be given a **procedural interpretation**. If we write a Horn clause in reverse order

b $\leftarrow$ a_1 and a_2 and a_3 . . . and a_n

we can view this as a definition of procedure b: the body of this procedure is given by the body of the clause (i.e. the operations indicated by the a's).

Prolog

The language gradually evolved in the 70's, mainly at the University of Aix-Marseille in France (Colmerauer) and in Edinburgh in Scotland (Kowalski), as an experimental artificial intelligence tool. Although there has been a number of dialects over the years the Edinburgh dialect has become a standard of sort. In 1981 Prolog received a major boost when the Japanese Institute for New Generation Computing Technology selected logic programming as its enabling software technology, however this did not have much success.

A Prolog program is a collection of Horn clauses defining relations. As a paradigm, it looks as follows:

1.	enter rules	(general truths)
2.	enter facts	(describe a particular situation)
3.	ask questions	(query)
4.	run time system searches for an answer.	

Values, variables, and terms

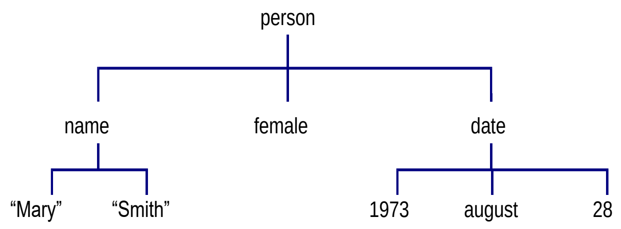
Prologs ***values*** are numbers, atoms, and structures. Lists are a particular kind of structure and strings are a particular kind of list.

Atoms: primitive values that have no properties other than the ability to distinguish one from another, used to represent real word objects that are primitive as far as the current application is concerned. For example: apple, purple, January, bob, mary. It is important to understand that Prolog ascribes no special meaning to individual atoms.

Structures: tagged tuples. For example, date (2000, january, 1) and date (2007, november, 22) might be used to represent dates where date represent a tag used to differentiate between two structures that happen to have the same components but represent distinct real-world objects (e.g. point (2, 3) and rational (2, 3)). The components of a structure can be any value, including sub-structures. Hence structures are hierarchical. For example:

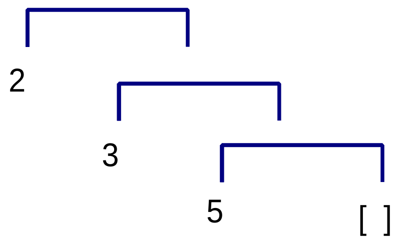
person (name (“Mary”, “Smith”), female, date (1973, august, 28))

corresponding to the following structure:



Lists subset of structures. The atom [ ] represents the empty list, and ‘[x | ys]’ represents the list whose head is x and whose tail is the list ys (i.e. [ys])

The list [2, 3, 5] is represented as:



Notice the similarity with car and cdr in Scheme.

A string is a list of integers (the internal representation of the characters in the string).

Prolog is an un-typed language. Numbers, atoms and structures can be used interchangeably (we can construct heterogeneous lists - we can also compare different sorts of value using “=” but such comparison will always yield *false*).

Term it is a constant (i.e. an atom or an integer), a variable, or a structure (structures represent the atomic propositions of predicate calculus, their general form is functor (parameter list) - functor are atoms used to identify the structure). Terms occur as arguments to relations. A simple assertion is of the form R (T_1, T_2, T_3, . . . , T_n) where each T_i is a term and R is the name of the relation.

Variable written in uppercase to distinguish it from atoms and tags, it denotes a fixed but unknown value of any type. (Note: Prolog variables correspond to mathematical variables, not the updatable variables of an imperative language). A variable is declared implicitly by its occurrence in a clause, and its scope is just the clause in which it occurs.

Clauses and Relations

A prolog program defines a collection of relations. Each relation is defined by one or more clause, written as follows:

A_0 :- A_1, A_2, . . . , A_n (:- is the Prolog notation for $\leftarrow$)

In addition, facts are bodiless clauses and queries are headless clauses.

ancestor (X,Y) :- parent(X,Z), ancestor(Z, Y)	a clause
ancestor (X,X)..:-	a fact
:- ancestor(bob,mary)	a query

The scope of every relation is the entire program.

As we will see later one of the important tools in Prolog is to keep eliminating terms in clauses to eventually derive the empty clause (i.e. :-). To do so we will match terms

For example:

age (P, Y) matches age (david, 42) under the substitution of
david for P, and 42 for Y

let us now look informally at two examples.

Celestial Bodies Example:

We first define a unary relation *star* on celestial bodies. Each celestial body is represented by an atom (e.g. sun). This relation is defined by three clauses:

```
star ( sun ).
star ( sirius ).
star ( betelgeuse ).
```

Note that the ":-" has disappeared and there is a period to terminate each clause.

Let us now consider the queries:

1.	?- star (sun).	yields yes	(it matches the first fact)
2.	?- star (jupiter).	yields no	(there is no matching fact)

We now define another relation (binary this time) *orbits*, using seven clauses:

```
orbits ( mercury,    sun ).
orbits ( venus,      sun ).
orbits ( earth,      sun ).
orbits ( mars,       sun ).
orbits ( moon,       earth ).
orbits ( phobos,     mars ).
orbits ( deimos,     mars ).
```

Consider the following queries:

3.	?- orbits (mars, sun).	Yes	(matches a fact)
4.	?- orbits (moon, sun).	No	(no matching fact)
5.	?- orbits (phobos, B).	B = mars	(making B equal to mars matches a fact)
6.	?- orbits (B, mars).	B = phobos & B = deimos	
7.	?- orbits (B, venus).	no, since no value of B would make this succeed	

The following clause defines the unary relation *planet*:

```
planet ( B )  :-     orbits ( B, sun ).
```

i.e. a body is a planet if it orbits the sun.

Let us look at the following queries:

```
8.    ?- planet ( mars ).          yes        (mars orbits the sun)
9.    ?- planet ( P ).             P = mercury, P = venus, P = earth, P = mars
```

Another clause defines the relation *satellite*

```
satellite ( B )  :-   orbits ( B, P ), planet ( P ).
```

i.e. a body is a satellite if it orbits some body P and that same body P is a planet.

Let us look at some queries:

```
10.   ?- satellite ( phobos ).          yes        (phobos orbits mars)
11.   ?- satellite ( S ).               S = moon, S = phobos, S = deimos
```

Other clauses define the relation *solar*:

```
solar ( sun ).
solar ( B )   :-      planet ( B ).
solar ( B )  :-       satellite ( B ).
```

i.e. the sun is a member of the solar system; a body is a member of the solar system if it is a planet; or if it is a satellite.

Consider now the following queries:

```
12.   ?- solar ( sun ).             yes        (it matches a fact)
13.   ?- solar ( moon ).            yes        (moon is a satellite)
14.   ?- solar ( sirius ).          no
```

```
15.    ?- solar ( B ).                    B = sun, B = mercury,
                                          B = venus, B = earth, B = mars,
                                          B = moon, B = phobos,
                                          B = deimos
```

If we want all possible answers, we type a ";" and push return. If we are satisfied with the answers so far we can type return by itself and the search will stop.

Family Relationships Example:

```
person ( X )  :-    mother_of ( Y, X ).⌉
person ( Y )  :-    mother_of ( Y, X ) ⌋.        rules

mother_of ( mary, carmen ).           ⌉
mother_of ( mary, alexander ).        |        facts
mother_of ( mary, andre ).            ⌋
```

Consider the following queries:

```
16.    ?- person ( carmen ).             yes
17.    ?- person ( alexander ).          yes
18.    ?- person ( andre ).              yes
19.    ?- person ( mary ).               yes (proven 3 ways )
20.    ?- person ( herb ).               no
```

Some things to remember:

- **upper case** is used for variables, **lower case** for constants & names of predicate and functions.
- Prolog answers **all questions** (even meaningless one)
- **arithmetic operators** are built in + * - / mod = >= =< /= < > !
- **assignment**: is denoted by "**is**"
- for **"and"** use the *comma*, for **"or"** the *semi-colon*
- all parameters can be input or output
- there is an explicit output command "write" which we won't use much

Few more facts:

```
        female ( carmen ).
        male ( alexander ).
        male ( andre ).
        parent ( carmen, herb, mary )
        parent ( alexander, herb, mary ).
        parent ( andre, herb, mary ).
        sister ( X, Y )   :-  female ( X ), parent ( X, F, M ), parent ( Y, F, M ).
```

Consider the following queries:

```
        21.   ?- sister ( carmen, alexander ).          yes
        22.   ?- sister ( ( carmen, mary ).              no
        23.   ?- sister ( carmen, carmen ).              yes
```

To prevent the last erroneous answer, we can rewrite the last clause above as:

```
        sister ( X, Y )  :-  female ( X ), parent ( X, F, M ), parent ( Y, F, M ), X /= Y.
```

Prolog has an arithmetic evaluator and arithmetic terms can be written in either infix or prefix notation. For example:

```
                3 + 4          or             + ( 3, 4 )
```

Prolog cannot tell when to consider an expression as data or when to evaluate it. “is” will force evaluation. For example:

```
      1.   ?- write ( 3 + 5 ).                              3 + 5
      2.   ?- X  is  3 + 5,  write ( X ).                   X = 8
```

Two arithmetic terms may not be equal even though they have the same value

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

Now we will investigate **how Prolog answers queries**

In general, a query consists of one or more goal. Let us review how a goal can be expressed as a Horn clauses:

a fact:	mammal (human) **:-**	(the "**:-**" is often omitted)
a query/goal	**:-** mammal (human)	

Prolog Search Strategy: Resolution and Unification

Resolution: this is an inference rule for Horn clauses that is especially efficient. If we can find two clauses such that we can match the head of the first clause with one of the statements in the body of the other, then the first clause can be used to replace, in the second clause, its head by its body. This means if we have:

$\boldsymbol{a}$:- $a_1, a_2, \ldots, a_n$
b :- $b_1, b_2, \ldots, \boldsymbol{b_i}, \ldots, b_m$

and **b_i matches a**, then we can infer the clause:
b :- $b_1, \ldots, b_{i-1}, \boldsymbol{a_1, a_2, \ldots, a_n}, b_{i+1}, \ldots, b_m$

For example, if we have only one statement in the Horn clauses, as in the following:

a :- b
b :- c

we can infer: a :- c as we saw earlier

The way it is done is simple: combine the two clauses and eliminate from both sides the terms that appear on both sides of the new proposition. In other words:

a, b :- b, c eliminate b on both sides to get a :- c

For example:

```
        from         older ( mary, carmen )  :-  mother_of ( mary, carmen ).
                     wiser ( mary, carmen )  :-  older ( mary, carmen ),

        one can infer
                     wiser ( mary, carmen )  :-  mother_of ( mary, carmen ).
```

A critically important property of resolution is its ability to detect any inconsistencies in a given set of propositions. This allows resolution to be used to prove theorems

Now let us see how a logic programming system can treat a *goal or list of goals* as a *Horn clause without a head*. The system attempts to apply resolution by matching one of the goals in the body of the headless clause with the head of a known clause and then replace the matched goal with the body of that clause, creating a new list of goals, which it continues to modify in the same way. When a goal is a compound proposition each of the facts (structure) is called a **subgoal**. For example:

(:-) a

and a :- $a_1, a_2, \ldots, a_n$

the original goal a is replaced by (:-) $a_1, a_2, \ldots, a_n$

If the system succeeds eventually in eliminating all goals (i.e. deriving the empty Horn clause) then the original statement has been proved.

For example:

if we have the fact:

```
        mammal ( human )  :-
```

and one asks whether a human is a mammal:

```
        :-  mammal ( human )
```

Using resolution, the system combines the two clauses:

```
                    mammal ( human )  :-  mammal ( human )
```

If we cancel both sides we get: :- [i.e. an empty clause]
and the system has found that a human is a mammal and responds “yes”

Unification: process by which variables are instantiated, or allocated memory and assigned values, so that patterns match during resolution. The purpose is to make two terms “the same.”

Let us consider another previous example:

```
( 1 )      legs ( X, 2 )      :-    mammals ( X ), arms ( X, 2 ).
( 2 )      legs ( X, 4 )      :-    mammals ( X ), arms ( X, 0 ).
( 3 )      mammal ( horse ).
( 4 )      arms ( horse, 0 )
```

Let us consider the query

```
( 5 )      :-    legs ( horse, 4 )
```

We can apply the resolution principle using the second rule and combining (5) and (2):

```
legs ( X, 4 )  :-  legs ( horse, 4 ), mammals ( X ), arms ( X, 0)
```

but in order to cancel the statements involving the predicate legs from each side, we have to match the variable X to horse. If indeed we replace X by horse in the entire statement, we get:

```
legs ( horse, 4 )  :-  legs ( horse, 4 ), mammals ( horse ), arms ( horse, 0 )
```

then, as we cancel the matching subgoal we get

```
( 6 )      :-  mammal ( horse ), arms (horse, 0)
```

if we now apply the resolution principle to (3) and (6)we get:

mammal (horse) :- mammal (horse), arms (horse, 0)

leading to (7) :- arms (horse, 0)

Finally, if we now apply the resolution principle to (7) and (4), we get

arms (horse, 0) :- arms (horse, 0)

leading to **:-** the null clause!

Hence the original query [legs (horse, 4)] is true.

This illustrate the fact that when we apply resolution to derive goals and we need to match statements that contain variables we must set the variables equal to terms so that the statements become identical and can be cancelled from both sides. This is **unification.** Hence an algorithm for unification needs to be provided.

Equality
Before we talk algorithm let us examine equality which is the basic expression whose semantics is determined by unification: in Prolog the goal s = t attempts to unify the term s and t. If unification succeeds the goal succeeds, it fails otherwise.

Example of the effect of equality

?- me = me	yes	
?- me = you	no	
?- me = X	X = me	
?- f (a, X) = f (Y, b)	X = b,	Y = a
?- f (X) = g (X)	no	(the functors are different)
?- f (X) = f (a, b)	no	(the number of parameter is different)
? f (a, g (X)) = f (Y, b)	no	(parameters don't match)
? f (a, g (X)) = f (Y, g (b))	X = b	Y = a

Your Turn …

Given the following:
Given the following:

```
( 1 )   star ( sun ).
( 2 )   star ( sirius ).
( 3 )   star ( betelgeuse ).

( 4 )   orbits ( mercury,       sun ).
( 5 )   orbits ( venus,         sun ).
( 6 )   orbits ( earth,         sun ).
( 7 )   orbits ( mars,          sun ).
( 8 )   orbits ( moon,          earth ).
( 9 )   orbits ( phobos,        mars ).
(10)    orbits ( deimos,        mars ).

(11)    planet ( B ):-      orbits ( B, sun ).

(12)    satellite ( B )     :-      orbits ( B, P ), planet ( P ).

(13)    solar ( sun ).
(14)    solar ( B )         :-              planet ( B ).
(15)    solar ( B )         :-              satellite ( B ).
```

what would the following queries return? Explain how the resolution is applied

```
?- solar ( mars ).

?- solar ( phobos ).

?- solar ( mercury ).

?- solar ( B ).
```

Algorithm for Unification for Prolog

1. A **constant** unifies only with itself (“me = me” suceeds but “me = you” fails).

2. A **variable** that is uninstantiated unifies with anything and become instantiated to that thing.

3. A **structured** item (i.e. a function applied to arguments) unifies with another term only if it has the same function name and the same number of arguments, and the arguments can be unified recursively. [f (a, X) unifies with f (Y, b) by instantiating Y to a and X to b . However f(a) = g(a) is an error].

A special case of 2 is when two uninstantiated variables are unified:

```
?-    X = Y              will lead to   X = 23    Y = 23
```

where 23 corresponds to the internal memory location assigned for that variable. Hence unification results in *uninstantiated variables to become alias of each other*.

Prolog Search Strategy

Prolog applies resolution in a strictly linear fashion, replacing goals left to right and considering clauses in the database in top-to-bottom order. Subgoals are also considered immediately once they are set up. This can be viewed as a depth-first strategy on a tree of possible choices.

For example:
Given:

```
        ancestor ( X, Y )  :-  parent ( X, Z ), ancestor ( Z, Y )          ( 1 )
        ancestor ( X, X )                                                  ( 2 )
        parent ( amy, bob )                                                ( 3 )
```

and the query:

```
        ?- ancestor ( H, bob )
```

Prolog will, based on the previous discussion, consider the following:

```
**          :- ancestor ( H, bob )                                 (the query)
        &   ancestor ( X, Y )  :-  parent ( X, Z ), ancestor ( Z, Y )    ( 1 )
```

It will unify X = H & Y = bob, leading to

```
                :-  parent ( H, Z ), ancestor ( Z, bob )
```

this is then matched with parent (amy, bob) :- (3)

and it unifies H = amy & Z = bob leading to

```
*               :- ancestor ( bob, bob )      &  { H = amy }       ( 4 )
```

this is then first matched with

```
            ancestor ( X, Y ) :- parent ( X, Z ), ancestor ( Z, Y )    ( 1 )
```

unifying X = bob & Y = bob

leading to :- parent (bob, Z), ancestor (Z, bob) & { H = amy)

which cannot be further matched, hence failure

From there the search backtracks to * and matches

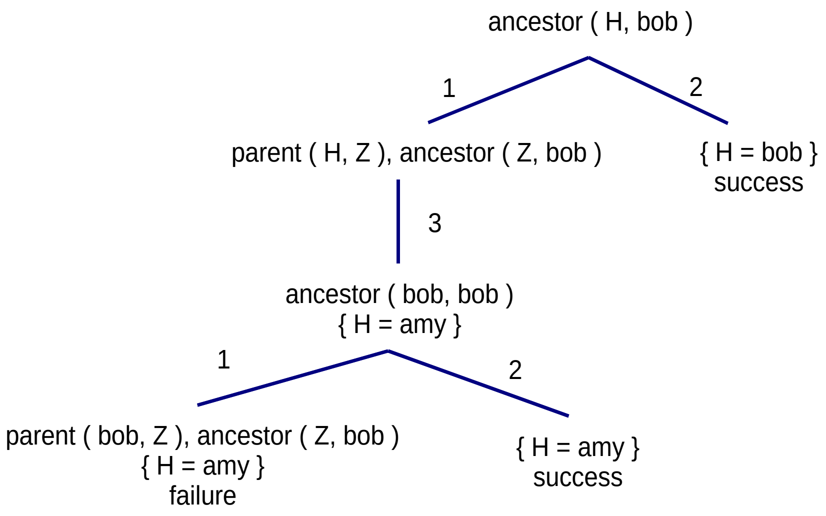
```
                :- ancestor ( bob, bob ) &  { H = amy }            ( 4 )
        &       ancestor ( X, X )                                  ( 2 )
```

it unifies   X = bob leading to the null clause and success with { H = amy }
Unless it is stopped it then backtracks to ** and matches

:- ancestor (H, bob)   [H = amy is released at this point]
& ancestor (X, X)   (2)

it unifies X = H & X = bob leading to the null clause and success with { H = bob }

The tree of subgoals can be viewed as follows (see the application of the resolution principle above):



(numbers on the lines refer to the clause above used by Prolog for resolution)

The depth-first approach is very efficient but can lead to infinite recursion if the tree has branches with infinite depth as would be the case if we re-write the previous rules as:

ancestor (X, Y) :- ancestor (Z, Y), parent (X, Z)	(1)
ancestor (X, X)	(2)
parent (amy, bob)	(3)

Prolog will go in an infinite loop trying to satisfy ancestor (Z, Y), continually re-using the first clause (this is caused by the left-recursive nature of that clause).

A breadth-first traversal would solve that problem but is much more expensive. *Prolog always uses depth-first*.

List Structures in Prolog

In addition to the **atomic proposition** (which looks like a function and its arguments), the structure we have encountered previously, Prolog supports the list which is a structure similar to that in LISP or Scheme.

In Prolog lists are delimited by square brackets and the elements are separated by commas. Lists are any sequence of any number of elements, where the elements can be atoms, atomic propositions, or any other term (atoms, variables, atomic proposition, or lists). For example:

    [apple, prune, grape, banana]

    [ ] represent the empty list.

Instead of having explicit functions for constructing or taking apart lists, Prolog uses the special notation

    [H | T]

to denote a list with head H and tail T. (*corresponding to car and cdr in Scheme*).

A list can be created with a simple structure:

someName ([apple, prune, grape, banana])

this indicates that the constant list [apple, prune, grape, banana] is a new element of the relation *someName*. It does not bind the list to *someName*, instead it states that [apple, prune, grape, banana] is a new element of *someName* in the same way as

    human (peter)

makes peter a new element of human. Hence we could also have:

someName ([apricot, peach, pear])

In query mode, the list can be dismantled into head and tail as in:

someName ([The_Head | The_Tail])

this will instantiate The_Head to apple, and The_Tail to [prune, grape, banana].

If backtracking caused a reevaluation (or if the two rules are switched) then The_Head and The_Tail would be reinstantiated respectively to apricot, and [peach, pear], since [apricot, peach, pear] is the next element of the relation *someName*.

The same notation used to dismantle the list can be used to construct one from given and instantiated, head and tail components. For example, if FirstElement is instantiated to strawberry and RestOfList is instantiated to [apricot, plum, raspberry] then

[First_Element | Rest_Of_List]

will create for this reference the list [strawberry, apricot, plum, raspberry]

Note that:

```
[ apple, peach, pear ]
[ apple | [ peach, pear ] ]
[ apple, peach, pear | [ ] ]
[ apple , peach | [ pear ] ]
```

are all equivalent and correspond to the same list.

Note also that a list of term is permitted to the left of the vertical bar. For example:

[X, a, Y | T] matches any list with at least three elements whose second element is the atom a. X will match the first element, Y will match the third, and T matches the rest of the list after the third element (possibly the empty list).

Examples:

```
    example ( [1, 2, 3, 4}).
    ?-example ([X | Y ] )
        X = 1
        Y = [ 2, 3, 4 ]
```

Certain operations are necessary to manipulate lists as is the case in Scheme. The process uses recursion as it does in Scheme. However, the recursion is caused and controlled by the resolution process.

Finding the last element of a list
(the first parameter is the list, the second is the result)

last ([X], X)	the last element of a singleton list is its only element
last ([X | Y], Z) :- not (Y = [ ]), last (Y, Z)	unless the tail is the empty list, it will be the last of the tail

Note that reversing the order of the statements will also work.

How does it work?:

?- last ([1, 2, 3] , L) is the query written as:

:- last ([1, 2, 3] , L)

it is combined with

last ([X | Y], Z) :- not (Y = [ ]), last (Y, Z)

where *X unifies to 1, Y to [2, 3], Z to L*

giving: :- not ([2, 3] = [ ], last ([2, 3] , L)

this is then combined with

last ([X' | Y'], Z') :- not (Y' = [ ]), last (Y', Z')

where *X' unifies to 2, Y' to [3], Z' to L*

giving :- not ([3] = [ ]), last ([3], L)

Note the different variables used.

this is then combined with

last ([X''], X'')

where X'' *unifies to 3 & L* *(i.e., X'' = 3 = L*

resulting in: L = 3 & yes

while ?- last ([ ], X) results in no (cannot be matched)

Appending two lists

(the first two parameters represent the two lists to be appended, the third is the resulting list)

```
append ( [ ] , L, L )                                  terminating condition
append ( [ H | L1 ], L2, [ H | L3 ] ) :-  append ( L1, L2, L3 )
```

Note that the terminating proposition is placed first because we know that Prolog will match the two propositions in order, starting with the first.

?- append ([1, 2]), [3, 4] , R)

will give the result R = [1, 2, 3, 4] & yes

Let us examine how this result is obtained:

We will combine the following:

:- append ([1, 2], [3, 4], R)

& append ([H | L1], L2, [H | L3]) :- append (L1, L2, L3)

[unify H = 1 L1 = [2] L2 = [3, 4] *<u>[1 | L3] = R]</u>* +

giving :- append ([2], [3, 4], L3)

which is then combined with:

append ([H' | L1'], L2', [H' | L3']) :- append (L1', L2', L3')

[unify H' = 2 L1' = [ ] L2' = [3, 4] *<u>[2 | L3'] = L3]</u>* ++

giving :- append ([ ], [3, 4], L3')

which is then combined with:

append ([ ] , L, L)

[unify L = [3, 4] L = L3' (alias)]

giving **:-** the null clause, i.e. success

from L3' = [3, 4] we then backtrack to ++ which will give:

<u>[2 | L3']</u> = [2 | [3, 4]] = [2, 3, 4] = L3

and we will backtrack to + *<u>[1 | L3</u>] = [1 | [2, 3, 4]] = [1, 2, 3, 4] = R*

Returning the result R = [1, 2, 3, 4]

In addition to appending two lists, append can be used to divide a given list:

?-append (X, Y, [a, b, c, d]) will give the following results :

```
    X = [ ], Y = [ a, b, c, d ]
    X = [ a ], Y = [ b, c, d }
    X = [ a, b ], Y = [ c, d ]
    X = [ a, b, c ], Y = [ d ]
    X = [ a, b, c, d ], Y = [ ]
```

Your Turn:

1. Define a third predicate so that third(X, Y) says that Y is the third element of the list X. The predicate should fail if X has fewer than three elements. Hint: this can be expressed as a fact.

2. Define a firstPair predicate so that firstPair(X) succeeds if and only if X is a list of at least two elements, with the first element the same as the second element. Hint: this can be expressed as a fact.

3. Define the evenSize predicate so that evenSize(X) says that X is a list whose length is an even number. Hint: you do not need to compute the length of the list, or do any integer arithmetic).

The results are:

```
third([A |[B |[Y | _ ] ] ] , Y).          or                  third2([A, B, Y| _], Y).

firstPair( [X, X | _]).

evenSize([ ]).
evenSize([_, _ | L]) :- evenSize( L ).
```

Loops and Control Structures

Depth-first search with backtracking can be used to perform loops and repetitive searches. What we need to do is to force backtracking even when a solution is found. This is done with the built-in predicate **fail**.

Example:

```
printpieces ( L )  :-  append ( X, Y, L ), write ( X ), write ( Y ), nl, fail
```

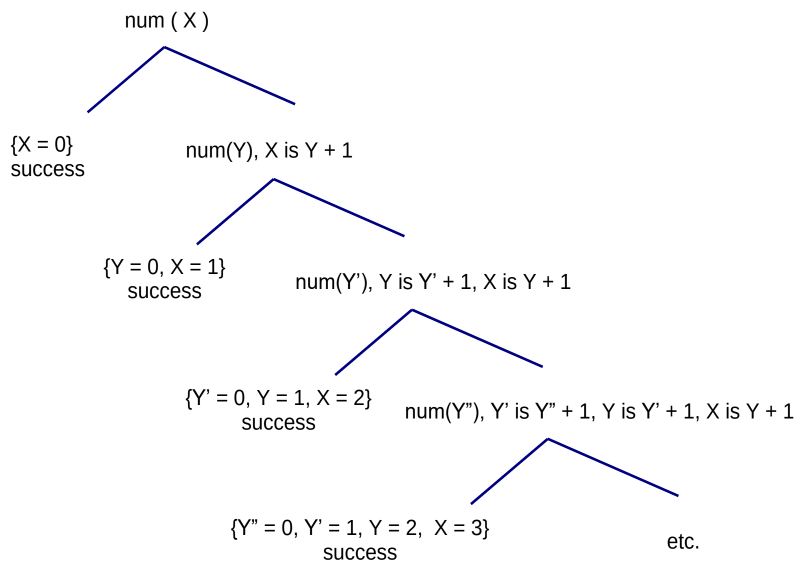
Hence we get:

```
?- printpieces ( [ 1, 2 ] )
[ ] [ 1, 2 ]
[ 1 ] [ 2 ]
[ 1, 2 ] [ ]
no
```

Forced backtracking on fail forces Prolog to write all solutions at once. This can be used to get repetitive computations. For example the following clauses:

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

the goal num (X) will produce all integers >= 0 as shown in the picture


num (X)
{X = 0}
success
num(Y), X is Y + 1
{Y = 0, X = 1}
success
num(Y'), Y is Y' + 1, X is Y + 1
{Y' = 0, Y = 1, X = 2}
success
num(Y"), Y' is Y" + 1, Y is Y' + 1, X is Y + 1
{Y" = 0, Y' = 1, Y = 2, X = 3}
success
etc.

We **could** try to generate all integers between 0 and 10 by writing:

```
writenum ( I, J )  :-  num ( X ),
                       I =< X,
                       X =< J,
                       write ( X ).
                       nl,
                       fail
```

and give the goal   writenum (1, 10)

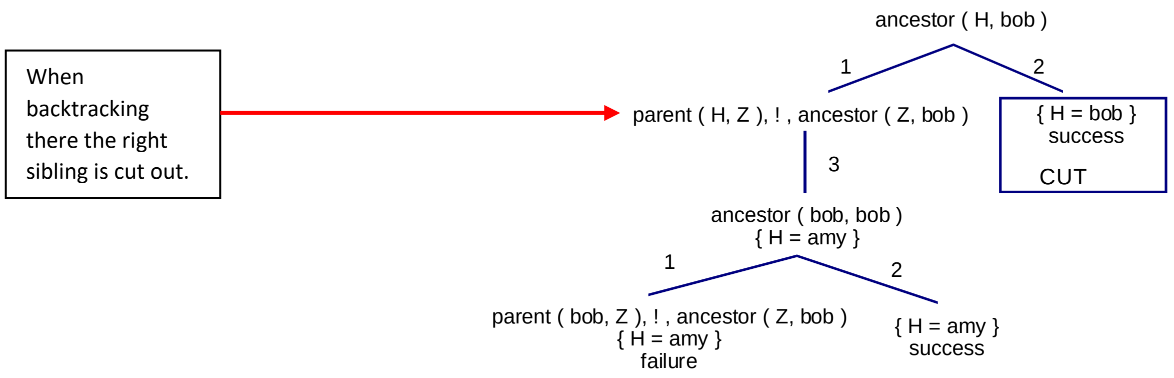
but this will go into an infinite loop after X = 10, generating ever increasing integers even though X =< 10 will never suceed. To address this problem Prolog has the **cut** operator, usually written as an exclamation point (**!**). The cut freezes the choice made when it is encountered. If a *cut is reached on backtracking, the search of the subtree of the **parent node of the node containing the cut** stops* (i.e. the siblings to the right of the node containing the cut), and the search continues with the grand-parent node (it prunes the search tree of all other siblings to the right of the node containing the cut. We will expand on this below.

CUT (denoted by !) and FAIL

Let us revisit the earlier example:

```
        ancestor ( X, Y )  :-  parent ( X, Z ), ! , ancestor ( Z, Y )        ( 1 )
        ancestor ( X, X )                                                    ( 2 )
        parent ( amy, bob )                                                  ( 3 )
```

As indicated previously, introducing a CUT we can control the results of a deep search as shown:

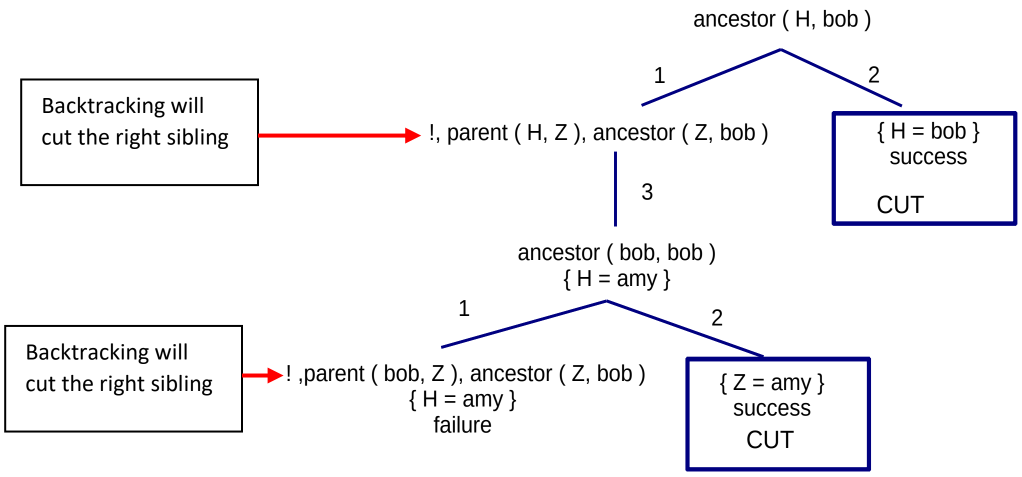


When backtracking there the right sibling is cut out.
ancestor (H, bob)
1
2
parent (H, Z), ! , ancestor (Z, bob)
{ H = bob } success CUT
3
ancestor (bob, bob) { H = amy }
1
2
parent (bob, Z), ! , ancestor (Z, bob) { H = amy } failure
{ H = amy } success

However, if we put the cut in the wrong place we could eliminate all solutions. For example:

```
        ancestor ( X, Y )  :-      !, parent ( X, Z ), ancestor ( Z, Y )        ( 1 )
```

will give the result below:



ancestor (H, bob)
1
2
Backtracking will cut the right sibling
!, parent (H, Z), ancestor (Z, bob)
{ H = bob } success CUT
3
ancestor (bob, bob) { H = amy }
1
2
Backtracking will cut the right sibling
! ,parent (bob, Z), ancestor (Z, bob) { H = amy } failure
{ Z = amy } success CUT

or eliminate no solution as in

```
ancestor ( X, Y )  :-      parent ( X, Z ), ancestor ( Z, Y )            ( 1 )
ancestor ( X, X )  :-      !.                                            ( 2 )
parent ( amy, bob )                                                      ( 3 )
```

Cut can be used to solve the problem of the infinite loop when printing the numbers between I and J

```
num ( 0 )
num ( X )    :-  num ( Y ), X is Y + 1
writenum ( I, J )  :-  num ( X ),
                       I =< X,
                       X =< J,
                       write ( X ). nl,
                       X = J, !.
                       fail
```

X = J will succeed when the upper bound J is reached and the cut will cause backtracking to fail, which stops the search for new values of X.

Another use of the cut is to imitate if-then-else constructs in imperative and functional languages.

To write a clause such as “D = if A then B else C” we write the following Prolog program:

D :- A, !. B.	if A is true, satisfy D with B, the cut prevents backtracking to try the next clause.
D :- C.	else (if A is false) satisfy D with C

Example:

```
        can_use ( Person, Facility )  :-  overdue ( Person, Book), ! , basic ( Facility )
        can_use ( Person, Facility )  :-  all (Facility )
        basic ( referencedesk )
        basic ( bathroom )
        all ( borrowing )
        all ( interlibraryloan )
        all ( F ) :- basic ( F )
        .
        .
        .
        overdue (scott, warnpeace )
        overdue ( mike, macbeth )
```

Check the result of the following query:

```
        ?-can_use (scott, borrowing).

        :-  can_use (scott, borrowing).
&       can_use ( Person, Facility )  :-  overdue ( Person, Book), ! , basic ( Facility )   **
        [ Person=scott, Facility=borrowing]

->      :- overdue ( scott, Book ), ! , basic ( borrowing )
&       overdue (scott, warnpeace )
        { Book = warnpeace }

->      !, basic ( borrowing )
        fails & returns to **
        no
```

Generate and Test

A very common design methodology for Prolog programs is called generate-and-test. Such a program includes both a mechanism for generating combinations of values, and another mechanism for testing to see if those values comprise a solution. It then relies on Prolog's search strategy to search all possible combinations of potential solutions.

Consider for example the problem of finding "Pythagorean Triples". These are values X, Y, Z, such that $X^2 + Y^2 = Z^2$. It is pretty easy to write a Prolog program that can "test" three values to see if they are a triple:

```
triple(X,Y,Z)     :-     W is X*X+Y*Y, V is Z*Z, W=V.
```

We can query this rule with triple(3,4,5) and it will properly return true. But querying it with something like triple(A,B,C), in the hopes that it finds triples, fails – because it cannot compute W when X and Y are uninitialized.

We need to add code that "generates" values for X, Y, and Z.

One very simple way of generating values is to just state the possible values as facts, as follows:

```
num(1).
num(2).
num(3).
etc.
```

The trick now is to combine this with the "test" code we just saw, preceded by bindings for X, Y, and Z using calls to "num":

```
num(1).
num(2).
num(3).
num(4).
num(5).
num(6).
num(7).
num(8).
num(9).
num(10).
```

```
triple(X,Y,Z) :- num(X), num(Y), num(Z),
W is X*X+Y*Y, V is Z*Z, W=V.
```

Now when we submit a query such as triple(A,B,C), it first tries to prove num(A). It satisfies this when it finds the rule num(1), thus setting A to 1. It does the same thing for Y and Z, and thus the first triple it "tests", is the triple (1,1,1). This of course, fails the test. So it backtracks, binding Z to 2. It will test (1,1,2), then (1,1,3), and so on, eventually finding (3,4,5) which passes. And it will continue, finding all triples for values between 1 and 10.

Of course, our program will become unwieldy if, for example, we wanted to find all triples for values in the range 1 to 1000

Generate-and-test can be used to solve problems without actually having to solve them ourselves. Imagine we wanted to sort a list of numbers, but we don't know any sorting algorithms. If we had a way of testing to see if a list is sorted, and a way of generating permutations of our list, we can use generate-and-test to have Prolog search for a sorted version. For example:

```
generate    test
     sort(S,T)           :-    permute(S,T), sorted(T).
     permute([],[]).
     permute(X,[Y|Z])    :-    concat(U,[Y|V],X),concat(U,V,W),permute(W,Z).
     sorted([]).
     sorted([X]).
     sorted([X,Y|Z])     :-    X=<Y, sorted([Y|Z]).
```

In this case, writing a Prolog program to find permutations of a list is arguably just as difficult as writing a sort algorithm! However, the concept of programming by coding how to recognize the answer, rather than how to find the answer, is partly what gives logic programming its appeal. Unfortunately, it also gives rise to one of *Prolog's drawbacks*. Although the above sorting algorithm works, it suffers from *terrible performance*. This is because it works by generating every possible permutation of the list, resulting in performance far worse even than a naive bubble sort.

[We won't most likely cover what follows but you might find it interesting]

Deficiencies of Prolog and Logic Programming Languages

There are several problems that arise when using Prolog as a logic programming language. Although it is a useful tool, it is neither a pure nor a perfect logic programming language.

Resolution Order Control: in a pure logic programming language the order of attempted matches during resolution is unspecified, all matches could be attempted concurrently. For reasons of efficiency Prolog specifies the order in which matches will be performed, thereby allowing the programmer to control the order of matching. This can have several negative results:

- slow execution if the programmer does not order the rules such as the most likely to succeed ones are first in the database.
- infinite loop (left recursive rules cause this) - a simple change of term ordering should not be crucial to the correctness of the program.
- explicit control of backtracking, via cut which is actually a goal, not an operator, it always succeed immediately but cannot be resatisfied through backtracking, prevents goals to the left of the cut from being resatisfied through backtracking. For example:

in the goal	a, b, !, c, d	if a & b succeed but c fails, the whole goal fails. This goal should be used only if it is known that whenever c fails it is a waste of time to resatisfy b or a.

For example:

```
member ( Element, [ Element | _ ] ).
member ( Element, [ _ | List ] )  :-  member ( Element, List ).
```

If the argument of member represents a set we know that member can only be satisfied once. Hence if member is a subgoal in a multi-goal statement, there can be a problem: if member succeed but the next goal fails, backtracking will try to resatisfy member by continuing a prior match. However, since the list argument of member has only one instance of the element member cannot succeed again and the whole goal will fail. Nevertheless Prolog will search the rest of the list wasting time. A solution is to use a cut to prevent further searching:

```
member ( Element, [ Element | _ ] ) :- !.
```

which will prevent backtracking from trying to resatisfy member and cause the entire goal to fail.

Closed-World Assumption

Prolog can create misleading results. As far as Prolog is concern the only truths are those in its database, it does not know anything about the world in general. Hence if it is given a query for which it has insufficient information to prove it to be true it will assume it is false.

Prolog can prove that a given goal is true but it cannot prove that a given goal is false. So if it cannot prove a goal true it will say it must be false. This is related to the negation problem.

```
human ( bob )
animal ( rover )

?- human ( X )
X = bob
?- not (human ( X ) ).
no                                  it does not know rover is not human.
```

The Negation Problem

Prolog has difficulty with negation. We have already seen the problem with.:

```
parent ( bill, jake ).
parent (bill, shelley ).
```

```
sibling (X, Y )  :-  parent (M, X), parent (M, Y).

?- sibling (X, Y)
```

where Prolog will respond in part with

```
X = jake
Y = jake
```

This occurs because the system will first instantiate M with bill and X with jake to satisfy the first subgoal. It will then start at the beginning of the database again to satisfy parent (bill, Y) and will instantiate Y with jake. This happens because the two subgoals are satisfied independently and the matching starts at the beginning of the database in both instances. As we have seen before we can solve this problem by specifying that to be a sibling of Y one must have the same parents and not be Y. Hence change the rule to read:

```
sibling (X, Y )  :-  parent (M, X), parent (M, Y), not (X = Y).
```

However, in other situations the solution may not be so simple.

Note that the "not" operator is satisfied if resolution cannot satisfy the subgoal X = Y. What this means is that we **are not guaranteed that X is not equal to Y but simply that resolution cannot prove from the database that X is equal to Y**. Hence the Prolog operator NOT is not equivalent to the logical operator NOT which means that its operand is provably false. This leads to a problem when we have a goal such as:

```
not ( not (some_goal)).
```

which should be equivalent to

```
Some_goal.
```

Consider the following:

```
member ( Element, [ Element | _ ] )     :-      !.
member ( Element, [_ | List ])      :-      member ( Element, List).
```

to discover one of the elements of a given list we could use the goal:

```
?- member ( X, [ mary, fred, barb ] ) .
```

which would instantiate X with mary. However, if we use:

```
not ( not ( member ( X, [ mary, fred, barb ] ) ) ).
```

which should give the same result but, in fact would return a non instantiated X. Why?

First the inner goal would succeed instantiating X to mary. Then Prolog would try to satisfy the next goal:

```
not ( member ( mary, [ mary, fred, barb ] ) )
```

which would fail because member succeeded. At this point X would be uninstantiated because Prolog always uninstatiates all variables in all goals that fail. After that Prolog would try to satisfy the outer “not” goal which would succeed because the inner “not” failed. Finally the result which is X would be printed but X is not instatiated at this point and such variable are usually printed as _n where n represents some string of digits. This is related to the fundamental reason is the form of a Horn clause:

$$A \ :- \ B_1 \wedge B_2 \wedge . . . \wedge B_n$$

If all the B propositions are true we can conclude that A is true but, regardless of the truth or falseness of the Bs it cannot be concluded that B is false. From positive logic one can only conclude positive logic. Hence the use of Horn’s clauses prevents any negative conclusion

Intrinsic Limitations

A fundamental goal of logic programming is to provide non-procedural programming. A system by which the programmer will specify what a program is supposed to do but not how this is to be done.
For example, as we saw earlier, the concept of sorting the elements of a list called old_list and placing them into a new list called new_list can be represented as

```
    sort (old_list, new_list)      ←      permute (old_list, new_list) ∧ sorted
                                                                  (new_list)
    sorted (list)                  ←  ∀ j    such that 1 ≤ j < n, list (j) ≤ list (j + 1)
```

where permute is a predicate which returns true if its second parameter is a permutation of its second parameter.

This can be rewritten in Prolog as:

```
sort ( S, T ) :-      permutation ( S, T), sorted ( T ).

permutation ( [ ], [ ] ).
permutation ( X, [ Y | Z ] )    :-  append ( U, [ Y | V ] , X),
                                    append ( U, V, W ),
                                    permutation ( W, Z ).

sorted ( [ ] ).
sorted ( [ X ] ).
sorted ( [ X, Y | Z] )  :-  X <= Y, sorted ( [ Y | Z ] )
```

The problem is that the sort algorithm will try all permutations until it finds the sorted one. It has no idea how to sort. So far no one has found a process by which the description of a sorted list can be transformed into some efficient algorithm for sorting.

A Challenging Problem: The Wolf, the Goat, the Cabbage Puzzle:

The problem is stated as such: A man (M) has got a wolf (W), a goat (Z) and a cabbage (K). He wants to cross a river in his boat, which is so small that he can either go over alone (M) or carry at most one of the (W,Z,K) with him. Naturally, when unobserved, the wolf will eat the goat, and the goat will eat the cabbage. Is there a sequence of boat movements that allows the man to transfer all three items across the river?

We will assume that initially all involved are on the west bank of the river and that the goal is to have them all on the east bank at the end.

We will represent the current configuration as[man, wolf, goat, cabbage] where each position can take the value w (on the west bank), or e (on the east bank). Initially the configuration will be [w, w, w, w], the desired final configuration is [e, e, e, e].

If, at first, the man would take the wolf across, the resulting configuration would be [e, e. w. w] and the goat would eat the cabbage.

In each move, the man crosses the river with either the wolf, the goat, the cabbage, or nothing. Each move can be represented with a corresponding atom: wolf, goat, cabbage, and nothing.
Now, each move transforms one configuration into another. This can be written as a Prolog predicate move(Config, Move, NextConfig) where: Config is a configuration, Move is one of the four basic moves (i.e. wolf, goat, cabbage, or nothing), and NextConfig is the configuration that results from applying that move to Config.

For instance, the goal move([w,w,w,w],wolf,[e,e,w,w]) would be valid and could be encoded as a fact:

```
move([w,w,w,w],wolf,[e,e,w,w]).      (note the '.')
```

But there would be MANY such facts to encode. The key observation here is that when the wolf and the man move, the goat and the cabbage do not change position. So a more general fact is:

```
move([w,w,Goat,Cabbage],wolf,[e,e,Goat,Cabbage]).
```

where Goat and Cabbage are variables equal to either w or e.

Now, there is a similar move when the man and the wolf go from East to West. One can thus further generalize the above fact as:

```
move([X,X,Goat,Cabbage],wolf,[Y,Y,Goat,Cabbage]) :- change(X,Y).
```

which assumes that a change predicate is defined as:

```
change(e,w).
change(w,e).
```

One could have thought of just writing:

move([X,X,Goat,Cabbage],wolf,[Y,Y,Goat,Cabbage]). but in this case X and Y above could unify to any atom (e.g. to goat), which is invalid.

Now, one can encode all the valid moves:

```
change(e,w).
change(w,e).

move([X,X,Goat,Cabbage],wolf,[Y,Y,Goat,Cabbage]) :- change(X,Y).
move([X,Wolf,X,Cabbage],goat,[Y,Wolf,Y,Cabbage]) :- change(X,Y).
move([X,Wolf,Goat,X],cabbage,[Y,Wolf,Goat,Y]) :- change(X,Y).
move([X,Wolf,Goat,Cabbage],nothing,[Y,Wolf,Goat,Cabbage]) :- change(X,Y).
```

Next, configurations need to be tested for safety (so that nothing gets eaten). To do this we define a predicate oneEq(X,Y,Z) that is true if at least one of Y or Z is equal to X.

```
    oneEq(X,X,_).
    oneEq(X,_,X).
```

The idea is that if at least one of the goat or the wolf is on the same bank as the man, AND if at least one of the goat or cabbage is on the same side as the man, then the configuration is safe (think about it for a minute). This can be encoded as:

```
        safe([Man,Wolf,Goat,Cabbage]) :-   oneEq(Man,Goat,Wolf),
oneEq(Man,Goat,Cabbage).
```

With move and safe, the puzzle can be solved. A solution is defined as a starting configuration and a list of moves that takes you to the goal configuration. A solution to [e,e,e,e] would be the empty list (no moves are needed). Otherwise, a solution is defined recursively as one move that takes you to a safe configuration followed by a solution. This recursion is easily encoded as:

```
    solution([e,e,e,e],[]).
    solution(Config,[FirstMove|OtherMoves]) :-  move(Config,Move,NextConfig),
        safe(NextConfig),
                        solution(NextConfig,OtherMoves).
```

WARNING: nothing in this Prolog program specifies how long the solution has to be. In fact, a solution could be arbitrary long (e.g. insert an infinite number of nothing moves when the goat is on one side and the wolf and the cabbage on the other). But it Prolog is asked for a solution of a specific length, is will oblige:

```
            ?- length(X, 7), solution([w, w, w, w],X).

    X = [goat, nothing, wolf, goat, cabbage, nothing, goat]

    Yes
```

which means: "First cross with the goat, then cross back empty, then cross with the wolf, then cross back with the goat, then cross with the cabbage, then cross back empty, a finally cross with the goat". It turns out that this is the minimum number of steps. If you keep asking Prolog for solutions you'll find that there are no solutions with an even number of steps, that some queries answer several times the same solution, etc..

The complete data base is:

```
change(e,w).
change(w,e).

move([X,X,Goat,Cabbage],wolf,[Y,Y,Goat,Cabbage]) :-    change(X,Y).
move([X,Wolf,X,Cabbage],goat,[Y,Wolf,Y,Cabbage]) :-    change(X,Y).
move([X,Wolf,Goat,X],cabbage,[Y,Wolf,Goat,Y]) :-    change(X,Y).
move([X,Wolf,Goat,C],nothing,[Y,Wolf,Goat,C]) :-    change(X,Y).

oneEq(X,X,_).
oneEq(X,_,X).

safe([Man,Wolf,Goat,Cabbage]) :-    oneEq(Man,Goat,Wolf),
oneEq(Man,Goat,Cabbage).

solution([e,e,e,e],[]).
solution(Config,[Move|Rest]) :-   move(Config,Move,NextConfig), safe(NextConfig),
solution(NextConfig,Rest).
```