Chapter 16

Logic Programming

Topics

- Introduction
- Predicate Calculus
- Propositions
- Clausal Form
- Horn Clauses

Logic Programming Paradigm

- AKA Declarative Paradigm
  - The programmer
    - Declares the goal of the computation (specification of results are stated).
    - Does not declare a detailed algorithm by which these goals are to be achieved.
- Application domain
  - Database design
  - Natural language processing
  - Artificial Intelligence
    - Automatic theorem proving
- Example language: Prolog

Logic Programming

- Instead of providing implementation, execute specification.
  - Relieves the programmer of specifying the implementation.
  - Express programs in a form of symbolic logic.
- Declarative specification:
  - Given an element x and a list L, to prove that x is in L, proceed as follows:
    - Prove that L is [x].
    - Otherwise, split L into L_1 and L_2 and prove one of the following:
      - x is in L_1 or
      - x is in L_2

Logic Programming

- Less effort to write, but implementation may be very inefficient.
  - Requires that the execution engine be more complex.
  - Use a logical inferencing (INFERENCE ENGINE) process to produce results

Introduction to Predicate Calculus

- Symbolic logic can be used for the basic needs of formal logic:
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions
- Particular form of symbolic logic used for logic programming is called *first-order predicate calculus*

Introduction to Predicate Calculus

- **Proposition**: a logical statement that may or may not be true.
  - Consists of objects and relationships of objects to each other.
    - Can either assert truth ("john speaks Russian") or query existing knowledge base ("does john speak Russian").
    - Can contain variables, which can become bound speaks(x,Russian).

Introduction to Predicate Calculus

- Example (English statements – Predicate Calculus)
  - 0 is a natural number
    - `natural(0).`
  - 2 is a natural number
    - `natural(2).`
  - For all x, if x is a natural number, then so is the successor of x.
    - `For all x, natural(x) → natural(successor(x)).`
  - -1 is a natural number
    - `natural(-1).`

Introduction to Predicate Calculus

- First and third logical statements are axioms for the natural numbers.
  - Statements that are assumed to be true and from which all true statements about natural numbers can be proved.
- Second logical statement can be proved from the previous axioms.
  - `2 = successor(successor(0)).`
  - `natural(0) → natural(successor(successor(0))).`
- Fourth logical statement cannot be proved from the axioms and so can be assumed to be false.

Predicate Calculus: statements

- Predicate calculus classifies the different parts of statements as:
  1. ***Constants***. These are usually number or names. Sometimes they are called *atoms*, since they cannot be broken down into subparts.
     - Example: `1, 0, true, false`
  2. **Predicates**. These are names for functions that are true or false, like Boolean functions in a program.
     - Can take any number of arguments.
     - Example: the predicate `natural` takes one argument.

Predicate Calculus: statements

3. ***Functions***. Predicate calculus distinguishes between functions that are true or false – these are predicates – and all other functions, which represent non-Boolean values.
   - Example: `successor`
4. **Variables**. Variables stand for as yet unspecified quantities.
   - Example: x
5. **Connectives**. These include the operations `and`, `or`, and `not`, just like the operations on Boolean data in programming languages. Additional connectives are implication and equivalence

Predicate Calculus: table of connectives

Name	Symbol	Example	Meaning	Notes
Negation	¬	¬a	not a	*True* if a is false; otherwise *false*
Conjunction	∩	a ∩ b	a and b	*True* if a and b are both true
Disjunction	∪	a ∪ b	a or b	*True* if either a or b (or both) is true
Equivalence	≡	a ≡ b	a is equivalent to b	*True* if a and b are both true or both false
Implication	⊃ ⊂	a ⊃ b a ⊂ b	a implies b b implies a	Logically equivalent to ¬a∪b

Predicate Calculus: connectives

- By convention, negation has highest precedence. Conjunctions, disjunctions, and equivalence have higher precedence than implication (in that order).
  - Example: $p \cup q \cap r \supset \neg s \cup t$ is equivalent to
    $((p \cup (q \cap r)) \supset ((\neg s) \cup t))$

Predicate Calculus: quantifiers

6. ***Quantifiers***. These are operations that introduce variables.
   - **Universal Quantifier:** `“for all”`
   - **Existential Quantifier**: `“there exists”`
   - A variable introduced by a quantifier is said to be **bound** by the quantifier.
   - It is possible for variables also to be **free** (not bound by any quantifier).
   - Quantifiers have higher precedence than any of the operators.

Predicate Calculus: table of quantifiers

Name	Symbol	Example	Meaning
Universal	$\forall$	$\forall$ X P	For all X, P is true
Existential	$\exists$	$\exists$ X P	There exists a value of X such that P is true

Predicate Calculus: quantifiers

- Examples:
  - ∀x(speaks(x,Russian))
    - True if everyone on the planet speaks Russian; false otherwise.
  - ∃x(speaks(x,Russian))
    - True if at least one person on the planet speaks Russian; false otherwise.
  - ∀x ∃y(speaks(x,y))
    - True if every person x speaks some language y; false otherwise.
  - ∃ x ∀ y(speaks(x,y))
    - True if at least one person on the planet speaks every language y; false otherwise.

Predicate Calculus: statements

7. ***Punctuation Symbols***. These include left and right parentheses the coma, and the period. Parentheses can be left out based on common conventions about the precedence of connectives.

- Arguments to predicates and functions can only be terms, that is, combinations of variables, constants, and functions. Terms cannot contain predicates, quantifiers, or connectives.

Predicate Calculus: examples

- prime(n)
  - *True* if the integer value of n is a prime number.
- 0 ≤ x + y
  - *True* if the real sum of x and y is nonnegative.
- speaks(x,y)
  - *True* if the person x speaks language y.
- 0 ≤ x ∩ + x ≤ 1
  - *True* if x is between 0 and 1, inclusive.
- speaks(x,Russian)∩speaks(y,Russian) ⊃ talkswith(x,y)
  - *True* if the fact that both x and y speak Russian implies that x talks with y
- ∀x(¬literate(x) ⊃ (¬writes(x)∩ ∃y(reads(x,y)∩ book(y))))
  - *True* if every illiterate person x does not write and has not read any book y.

Predicate Calculus: tautologies

- *Tautologies*: Propositions that are true for all possible values of their variables.
  - Example: q ∪ ¬q
- Predicates that are *true* for some particular assignment of values to their variables are called *satisfiable*.
  - Example: `speaks(x,Russian)`
    - If at least one person in the planet speaks Russian.
- Predicates that are *true* for all possible assignments of values to their variables are *valid*.
  - Example: `even(y) ∪ odd(y)`
    - It is true for all integers

Propositions: Summary

- Objects in propositions are represented by simple terms: either constants or variables
- ***Constant***: a symbol that represents an object
- ***Variable***: a symbol that can represent different objects at different times
  - Different from variables in imperative languages

Propositions: Summary

- Simplest propositions are called *atomic propositions* which consist of compound terms
- A *compound term* is composed of two parts
  - *Functor*: function symbol that names the relationship.
  - Ordered list of *parameters* (tuple)

Propositions: Summary

- Examples:
```
student(jon)
like(seth, OSX)
like(nick, windows)
like(jim, linux)
```
- Propositions can be stated in two forms:
  - *Fact*: proposition is assumed to be true
  - *Query*: truth of proposition is to be determined
- Compound proposition:
  - Have two or more atomic propositions
  - Propositions are connected by operators

Clausal Form

- Problem of predicate calculus:
  - Too many ways to state the same thing
- Solution: use a standard form for propositions
- All propositions can be expressed in ***clausal form***:

$$B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$$

  - means if all the As are *true*, then at least one B is *true*
- Characteristics of clausal form:
  - Existential quantifiers are not required.
  - Universal quantifiers are implicit with use of variables.
  - No operator other than conjunctions and disjunctions.

Clausal Form

- *Antecedent*: right side of proposition.
- *Consequent*: left side of the proposition.

$$\underbrace{\texttt{likes(bob,mary)}}_{\text{consequent}} \subset \underbrace{\texttt{likes(bob,redheads)} \cap \texttt{redhead(mary)}}_{\text{antecedent}}.$$

- A proposition with zero or one terms in the consequent is called a *Horn clause*.

Horn Clauses

- A Horn clause has a head h, which is a predicate, and a body, which is a list of predicates $p_1, p_2, \ldots p_n$

$$\underbrace{p_1, p_2, \ldots p_n}_{\text{body}} \rightarrow \underbrace{h}_{\text{head}}$$

  - In a Horn clause the head is true if every predicate of the body is true (simultaneously).

Horn Clauses: facts and queries

- *Fact*: a Horn clause without body.
  - They are called *headless* Horn clauses.
    - → h or just h
  - It means that h is always *true*.
    - Example: → `mammal(human).`
- *Query:* a Horn clause without a head.
  - The "opposite" of a fact.
    - Example: `mammal(human)` →

From Predicates to Horn Clauses

- There is a limited correspondence between Horn clauses and predicates.
  - Horn clauses can be written equivalently as a predicate
    - HC: `snowing(C)` ← `precipitation(C),freezing(C).`
    - PC: `snowing(C)` ⊂ `precipitation(C)`∩`freezing(C).`
  - Not all predicates can be translated into Horn clauses.

Properties of Predicate Logic Expressions

Property	**Meaning**	
Commutativity	$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
Associativity	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Distributivity	$p \vee q \wedge r \equiv (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Idempotence	$p \vee p \equiv p$	$p \wedge p \equiv p$
Identity	$p \vee \neg p \equiv true$	$p \wedge \neg p \equiv false$
deMorgan	$\neg (p \vee q) \equiv \neg p \wedge \neg q$	$\neg (p \wedge q) \equiv \neg p \vee \neg q$
Implication	$p \supset q \equiv \neg p \vee q$	
Quantification	$\neg \forall x\, P(x) \equiv \exists x\, \neg P(x)$	$\neg \exists x\, P(x) \equiv \forall x\, \neg P(x)$

From Predicates to Horn Clauses

- Six-step procedure that will, whenever possible, translate a predicate p into a Horn clause.

1. Eliminate implications from p, using the implication property.
2. Move negation inward in p, using the deMorgan and quantification properties, so that only individual terms are negated.
3. Eliminate existential quantifiers from p, using a technique called *skolemization*. Here, the existentially quantified variable is replaced by a unique constant.
   - For example, the expression ∃x`P(x)` is replaced by `P(c)`, where c is an arbitrarily chosen constant in the domain of `x`.

From Predicates to Horn Clauses

4. Move all universal quantifiers to the beginning of p; as long as there are no naming conflicts, this step does not change the meaning of p. Assuming that all variables are universally quantified, we can drop the quantifiers without changing the meaning of the predicates.
5. Use the distributive, associative, and commutative properties to convert p to *conjunctive normal form*. In this form, the conjunction and disjunction operators are nested no more than two level deep, with conjunctions at the highest level.

From Predicates to Horn Clauses

6. Convert the embedded disjunctions to implications, using the implication property. If each of these implications has a single term on its right, then each can be rewritten as a series of Horn clauses equivalent to p.

- Example:

`∀x(¬literate(x)⊃(¬writes(x)∧¬∃y(reads(x,y)∧book(y))))`

- Example:

`∀x(literate(x)⊃reads(x)∨writes(x))`

Topics

- Proving Theorems
  - Resolution
  - Instantiation and Unification
- Prolog
  - Terms
  - Clauses
  - Inference Process
  - Backtracking

Predicate Calculus and Proving Theorems

- A use of propositions is to discover new theorems that can be inferred from known axioms and theorems
- *Resolution*: the process of computing inferred propositions from given propositions
  - Resolution principle is similar to the idea of transitivity in algebra.

Resolution

- Making a single inference from a pair of Horn clauses.
  - If h is the head of a Horn clause and it matches with one of the terms of another Horn clause, then that term can be replaced by h.
    - The Horn clauses:

      $h \leftarrow terms$
      $t \leftarrow t_1,\ h,\ t_2$
    - The second clause is resolved to

      $t \leftarrow t_1,\ terms,\ t_2$

Resolution: example

- Consider the following clauses:

```
speaks(Mary,English).
talkswith(X,Y) ← speaks(X,L), speaks(Y,L), X≠Y
```

- Resolution allow us to deduce:

```
talkswith(Mary,Y) ← speaks(Mary,English),
                    speaks(Y,English), Mary≠Y
```

  - Variables X and L are assigned the values "Mary" and "English" in the second rule.
  - The assignment of values to values during resolution is called *instantiation*.

Unification and Instantiation

- *Unification*: finding values for variables in propositions that allows matching process to succeed
- *Instantiation*: assigning temporary values to variables to allow unification to succeed
- After instantiating a variable with a value, if matching fails, may need to backtrack and instantiate with a different value.

Resolution: Theorem Proving

- Use proof by contradiction.
- *Hypotheses*: a set of pertinent propositions
- *Goal*: negation of theorem stated as a proposition.
- Theorem is proved by finding an inconsistency.

The language Prolog

- The most widely used logic programming language.
- Prolog in a nutshell
  - Uses Horn clauses
    - Almost identical notation of Horn clauses, except the implication arrow "←" is replaced by a colon followed by a dash ":-".
  - Implements resolution using strict linear "depth first" strategy and a unification algorithm.

A Prolog Program

- Prolog approach.
  - Describe known facts and relationships.
- A program consists of a sequence of Horn clauses.
  - Clauses are implications of the form
    - p_1 if p_2 … & p_k written as p_1 :- p_2, … ,p_k where every p is a term.
      - p_1 is called the *head* and p_2 , … , p_k is the *body*.
      - The body is a list of *goals* separated by commas (conjunctions).

Terms

- All Prolog statements are constructed from terms.

$$\left.\begin{array}{l}\left.\begin{array}{l}\text{Atoms}\\ \text{Numbers}\end{array}\right\}\text{Constants}\\ \text{Variables}\end{array}\right\}\text{Simple Terms}$$

Compound/Complex terms (structures)

Terms: atoms

- Atoms are used as names of individuals and predicates.
- Atoms can be constructed in three ways:
  1. Strings of letters, digits and the underscore character, starting with a lower-case letter.
     - `x`
     - `vancouver`
     - `ax123aBCD`
     - `abc_123_etc`

Terms: atoms

2. Strings of special characters .
   - Some of these have a predefined meaning (:-).
   - `-->`
   - `==>`
3. String of characters enclosed in single quotes.
   - It can contain any character.
   - `'Florida'`
   - `'12$12$'`
   - `'  back\\slashes'`
   - `'32'` is an atom, not equal to the number 32.
   - A zero length atom is written ''

Terms: atoms

- Internal representation:
  - Atoms are not character strings, they are locations in a symbol table.
  - Only one copy of each atom is stored.
  - All occurrences are replaced by pointers to its location in the symbol table.
    - `abracadabraabracazam= abracadabraabracazam`
    - `a = b`
    - Comparison takes the same amount of time.
      - Pointers get compared rather than the strings of characters.

Terms: numbers

- Integer
  - Useful for task such as counting the elements of a list.
- Real
  - Not used very much in typical Prolog programming.

Terms: variables

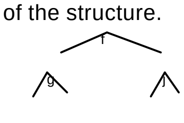
- Stand for objects that we cannot name.
- Begins with a capital letter or '_'.
  - `X`
  - `_value`
  - `Mother`
- A variable can be
  - *Instantiated* when there is an object that it stands for.
  - *Uninstantiated* when what the variable it stands for is not yet known.

Terms: variables

- Question containing a variable:
  - Search through all facts to find an object that the variable could stand for.
- Anonymous variable (_)
  - A special variable that matches anything, but never takes on a value.
  - Successive anonymous variables in the same clause do not take on the same value.

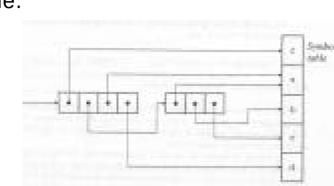
Terms: structures

- Consists of a *functor* followed by a sequence of *arguments*.
  - The functor must be an atom.
  - Arguments can be any kind of terms (including other structures).
  - The outermost functor (`f/2`, in this case) is called the *principal functor* of the structure.
    - `f(g(h,i),j(k,1))`



Terms: structures

- Internal representation
  - A structure is a linked tree made of pointers to its substructures and to entries in the symbol table.



Prolog Clauses

- Prolog clauses are of three types:
  - *Facts:* declare things that are always, unconditionally true.
  - *Rules:* declare things that are true, depending on a given condition.
  - By means of *questions* users can ask the program what things are true.

Prolog Clauses: characteristics

- A Prolog clause:
  - Arbitrary number of arguments (parameters).
  - A predicate that takes `N` arguments is called *N-placed predicate.*
  - A one-place predicate describes a *property* of one individual; a two-place predicate describes a *relation* between two individuals.
  - The number of arguments that a predicate takes is called its *arity.*

Prolog Clauses: characteristics

- Two distinct predicates can have the same name if they have different arities.
  - `mother(pam)` meaning Pam is a mother.
  - `mother(pam,bob)` meaning Pam is the mother of Bob.
- A predicate is identified by giving its name, a slash, and its arity.
  - `mother/1.`
  - `mother/2.`
- Every Prolog statement is terminated by a period.

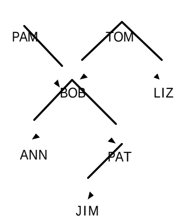
Facts

- Define relationships between objects.
- Sometimes called *ground clauses* because they are the basis from which other information is inferred.
- Facts are clauses that have an empty body.
- Facts are written as: `name_of_relationship(object,…,object).`
- A collection of facts and rules is called a *database* or *knowledge base.*

Facts: examples

```
parent(tom,bob).
parent(pam,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```

Family Relation



Rules

- Specify things that are true if some condition is satisfied.
- Rules are used to say that a fact depends on a group of other facts.
- A rules consists of a head and a body.
  - Connected by the symbol `':-'` (if).
  - The head describes what fact the rule is intended to define.
  - The body describes the conjunctions of goals that must be satisfied for the head to be true.

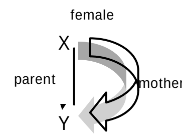
Rules: examples

For all X and Y,
Y is an offspring of X if
X is a parent of Y.

$\texttt{?-}\underbrace{\texttt{offspring(Y,X)}}_{\text{head/conclusion}}\texttt{:-}\underbrace{\texttt{parent(X,Y)}}_{\text{body/condition}}\texttt{.}$

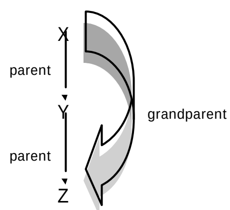
Rules: examples

```
?-mother(X,Y):- parent(X,Y),female(X)
```



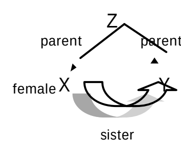
Rules: examples

```
?-grandparent(X,Z):- parent(X,Y),parent(Y,Z).
```



Rules: examples

```
?-sister(X,Y):- parent(Z,X), parent(Z,Y),
            female(X).
```



Rules: exercise

- Define the relation `aunt(X,Y)`

Questions

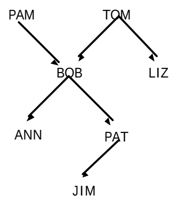
- How do we use Prolog programs?
  - By posing queries (a query is a request for the computer to do something.)
- Questions are clauses that only have the body.
  - Search through the database.
  - Look for facts that *match* the fact in question.
    - Two facts *match* if their predicates and corresponding arguments are the same.
      - YES: Prolog finds a match
      - No: Prolog does not find a match (nothing matches the question vs. false)

Questions: example

User Queries	Prolog's Answers
`?-parent(bob,pat).`	`yes`
`?-parent(liz,pat).`	`no`
`?-parent(tom,ben).`	`no`
`?-parent(X,liz).`	`X=tom`
`?-parent(bob,X).`	`X=ann` `X=pat`

`?-parent(X,Y).`	`X=pam`
	`Y=bob;`
	`X=tom`
	`Y=bob;`
	`...`

Family Relation



Conjunctions

- More complicated relationships.
- Satisfy two or more separate goals.
- Commas are understood as conjunctions.
- Question containing conjunctions:
  - Try to satisfy each goal in turn (searching the database).
  - All goals have to be satisfied.

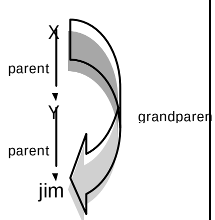
Complex Queries: example

Who is a grandfather of Jim?

1. Who is a parent of Jim?
   Assume that this is some Y.
2. Who is a parent of Y?
   Assume that this is some X.

```
?-parent(X,Y),parent(Y,jim).

X=bob
Y=pat
```



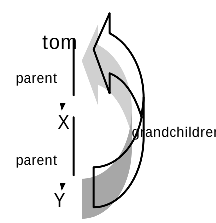
Complex Queries: example

Who are Tom's grandchildren?

1. Who is a child of Tom?
   Assume that this is some X.
2. Who is a child of X?
   Assume that this is some Y.

```
?-parent(tom,X),parent(X,Y).
X=bob
Y=ann

X=bob
Y=pat
```



Complex Queries: example

Do Ann and Pat have a common parent?

1. Who is a parent, X, of Ann?
2. Is (this same) X a parent of Pat?

```
?-parent(X,ann),parent(X,pat).
X=bob
```


Complex Queries: exercises

What will be Prolog's answers to the following questions?

1. `?-parent(jim,X).`
2. `?-parent(X,jim).`
3. `?-parent(pam,X), parent(X,pat).`
4. `?-parent(pam,X),parent(X,Y),parent(Y,jim).`

Formulate in Prolog the following questions:

1. Who is Pat's parent?
2. Does liz have a child?
3. Who is Pat's grandparent?

Matching / Unification

- Two terms (`term`$_1$ and `term`$_2$) can be unified (matched) if they are alike or can be made alike by instantiation.
  - Instantiation: make one variable the same as another.

Matching / Unification

(1) If `term`$_1$ and `term`$_2$ are constants
- Same *atom* or same *number*.

constants

(2) If `term`$_1$ is a *variable* and `term`$_2$ is any type of term
- `term`$_1$ is instantiated to `term`$_2$

(3) If `term`$_1$ and `term`$_2$ are *variables*
- They are instantiated to each other (share values).

variables

(4) If `term`$_1$ and `term`$_2$ are *complex terms*

a. They have the same functor and arity.
b. All their corresponding arguments match.
c. The variable instantiations are compatible.

complex terms

(5) Two terms match if and only if it follows from the previous four clauses that they match.

Unification: examples

User Queries	Prolog's Answers
`?-2=2.`	yes
`?-bob=jim.`	no
`?-'bob'=bob.`	yes
`?-'2'=2.`	no
`?-bob=X.`	X=bob yes
`?-X=Y.`	yes
`?-X=bob. X=jim.`	no
`?-kill(shoot(gun),Y))=kill(X,stab(knife)).`	X= shoot(gun) Y=stab(knife) yes
`?-kill(shoot(gun),stab(knife))=kill(X,stab(Y)).`	X= shoot(gun) Y=knife yes

Inference Process of Prolog

- Queries are called *goals*
- If a goal is a compound proposition, each of the facts is a *subgoal*
- To prove a goal is true, must find a chain of inference rules and/or facts. For goal Q:

```
B :- A
C :- B
…
Q :- P
```


Inference Process of Prolog

- Process of proving a subgoal is called *matching*, *satisfying*, or *resolution*.
- Bottom-up resolution, forward chaining
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
  - Works well with a large set of possibly correct answers
- Top-down resolution, backward chaining
  - Begin with goal and attempt to find sequence that leads to set of facts in database.

Inference Process of Prolog

  - works well with a small set of possibly correct answers
- Prolog implementations use backward chaining
- When goal has more than one subgoal, can use either
  - Depth-first search: find a complete proof for the first subgoal before working on others
  - Breadth-first search: work on all subgoals in parallel.
- Prolog uses depth-first search

A Simple Prolog Knowledge Base

- A Prolog knowledge base that describes the location of certain North American cities.

```
/* 1 */  located_in(atlanta,georgia).
/* 2 */  located_in(houston,texas).
/* 3 */  located_in(austin,texas).
/* 4 */  located_in(toronto,ontario).
/* 5 */  located_in(X,usa) :- located_in(X,georgia).
/* 6 */  located_in(X,usa) :- located_in(X,texas).
/* 7 */  located_in(X,canada) :- located_in(X,ontario).
/* 8 */  located_in(X,north_america) :-
                                  located_in(X,usa).
/* 9 */  located_in(X,north_america) :-
                                  located_in(X,canada).
```


Inference Process: example

- Query:

`?- located_in(austin,north_america).`

Unifies with the head of Clause 8 by instantiating X as `austin`.

The right-hand side of Clause 8 becomes the new goal.

Goal: `?- located_in(austin,north_america).`

Clause 8: `located_in(X,north_america) :- located_in(X,usa).`

Instantiation: `X = austin`

New goal: `?-located_in(austin,usa).`

Inference Process: example

Unify the new query with Clause 6:

Goal: `?- located_in(austin,usa).`

Clause 6: `located_in(X,usa) :- located_in(X,texas).`

Instantiation: `X = austin`

New goal: `?-located_in(austin,texas).`

This query matches Clause 3. Since Clause 3 does not contain an "if", no new query is generated and the process terminates successfully.

Backtracking

- If several rules can unify with a query, how does Prolog know which one to use?
- Prolog does not know in advance which clause will succeed but it does know how to black out of blind alleys.
- Prolog tries the rules in order in which they are given in the knowledge base.
  - If a rule does not lead to success, it backs up and tries another

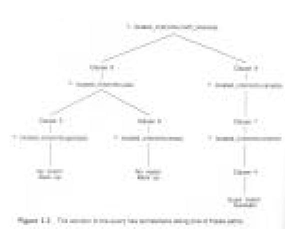
Backtracking

- The query `?-located_in(austin,usa)` will try to unify with Clause 5 and then, when that fails, the computer will back up and try Clause 6.
- A good way to conceive of backtracking is to arrange all possible paths of computation into a tree.
- Consider the query:

`?- located_in(toronto,north_america).`

Backtracking: example

- The following tree shows all the paths that the computation might follow.

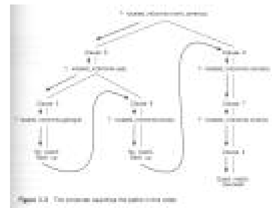


Backtracking: example

- We can prove that Toronto is in North America if we can prove that it is in either the U.S.A. or Canada.
- If we try the U.S.A., we have to try several states and then Canada.
- Almost all paths are blind alleys.
- Only the rightmost one leads to a successful solution.

Backtracking: example

- Same diagram with arrows added to show the order in which the possibilities are tried.



Backtracking: example

- Whenever the computer finds it has gone down a blind alley, it backs up to the most recent query for which there are still untried alternatives, and tries another path.
- When a successful answer is found, the process stops.

Prolog Syntax: comments

- Two ways to delimit comments
  - Anything bracketed by `/*` and `*/`
  - `/* This is a comment */`
- Anything between `%` and the end of the line
  - `% This is also a comment`

Sicstus

- http://www.cs.sfu.ca/CC/SW/Prolog/
- Linux and SunOS machines
  - CSIL SunOS machines have an additional Prolog implementation: BinProlog.
- Running Sicstus Prolog
  - *orion%* sicstus

Running Sicstus

- Interactive definition mode:
  - The special goal [user] is used to enter interactive definition mode.
  - In interactive mode, Prolog expects goals to establish.
  - ^D (i.e., the ctrl-D key) exits definition mode.
- Consulting a file
  - The special query

    `?-consult('family.pl').`

    asks prolog to read the definitions from the named file in quotes.
  - Goals must be terminated with a period or Prolog just waits until you enter the period.

Running Sicstus

- To load the same program use `reconsult` instead of `consult`.
  - Otherwise, there will be two copies of it in memory at the same time.
- To exit from Prolog just type the special query `?- halt.`
- If a single query has multiple solutions, Prolog finds one solution and then asks whether to look for another (until all alternatives are found or you stop asking for them).

Running Sicstus

```
?- located_in(X,texas).
       X = houston
       More (y/n) ? y
       X = austin
       More (y/n) ? Y
       no
```

- The *"no"* at the end means there are no more solutions.
- Any of the arguments of a predicate can be queried.
  - `?- located_in(austin,X).` % Names of regions that contain Austin
  - `?- located_in(X,texas).` % Names of cities that are in Texas
  - `?- located_in(X,Y).` % "What is in what?"
  - `?- located_in(X,X).` % "What is in itself?"

Sicstus: examples

- Sample Prolog session

```
1: orion% sicstus
SICStus 3.8.4 (sparc-solaris-5.6): Mon Jun 12 18:49:23 MET DST 2000
Licensed to cs.sfu.ca
| ?- [user].
| member(X, [X|_]).
| member(X, [_|More]) :- member(X, More).
| ^D
|{consulted user in module user, 0 msec 336 bytes}
| ?- ['data.pl'].
{consulting /cs/gard1/dma/family.pl...}
{consulted / cs/gard1/dma/family.pl in module user, 10 msec 160 bytes}
yes
| ?- parent(X,liz).
X = tom;
no
| ?- halt.
2: orion%
```


Chapter 16

Logic Programming

Topics

- Summary (resolution, unification, Prolog search strategy)
- Disjoint goals
- The "cut" operator
- Negative goals
- Predicate "fail"
- Debugger / tracer
- Arithmetic
- Lists

Resolution

- Resolution is an inference rule for Horn clauses.
- Given two clauses:
  - If the head of the first clause can be matched with one of the statements in the body of the second clause then the first clause can be used to replace its head in the second clause by its body.

$a \leftarrow a_1, \ldots, a_n.$
$b \leftarrow b_1, \ldots, b_i, \ldots, b_m.$

and b_i matches a, then

$b \leftarrow b_1, \ldots, b_{i-1}, a_1, \ldots, a_n, b_{i+1}, \ldots, b_m$

Unification

- Unification is the process by which variables are instantiated so that patterns match during resolution.
- Unification is the process of making two terms "the same" in some sense.
  - `'foo' = foo`
    - Prolog's answer: `yes`.
    - Both terms are atoms.
  - `'5' = 5`
    - Prolog's answer: `no`.
    - LHS is an atom and RHS is a number.

Prolog's Search Strategy

- Prolog applies resolution in a strictly linear fashion
  - Replaces goals left to right.
  - Considers clauses in top-to-bottom order.
  - Subgoals are considered immediately once they are set up.
  - Search can be viewed as a depth-first search on a tree of possible choices.

Prolog's Search Strategy

- Given the following clauses:

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

- Given the goal `ancestor(X,bob)`, Prolog's search strategy is left to right and depth first on the following tree of subgoals.
  - Edges are labelled by the number of the clause used by Prolog for resolution
  - Instantiation of variables are written in curly brackets

Prolog's Search Strategy

- Leaf nodes in this tree occur
  - No match is found for the leftmost clause.
  - All clauses have been eliminated (success).
- Whenever failure occurs Prolog backtracks up the tree to find further paths to a leaf, releasing instantiations of variables.

Prolog's Search Strategy

- Original Prolog program:

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

- Clauses in slightly different order:

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

- Problem?

Existential Queries

- Variables in queries are existentially quantified.

```
father(abraham,X)?
```

  - Reads: "Does there exist an `X` such that `abraham` is the father of `X`?"
  - For convenience, existential quantification is omitted.

Universal Facts

- Variables in facts are implicitly universally quantified.
- In general, a fact `p(T1,…Tn)` reads that for all $X_1, \ldots, X_k$ where the X_i are variables occurring in the fact $p(T_1, \ldots T_n)$ is true.

```
likes(X,apple).
```

- From a universally quantified fact one can deduce any instance of it.

```
likes(adan,apple).
```


"Universal" Rules

- Rule specifies things that are true if some condition is satisfied.

For all X and Y,
Y is an offspring of X if
X is a parent of Y.

```
?-offspring(Y,X):-parent(X,Y).
```


Disjoint Goals ("or")

- Prolog provides a semicolon, meaning "or".
  - The definition of `parent` could be written as a single rule:

```
parent(X,Y) :- father(X,Y); mother(X,Y).
```

- The normal way to express an "or" relation in Prolog is to state two rules.
  - The semicolon adds little or no expressive power to the language.
  - It looks so much like the comma that it often leads to typographical errors.
  - The use of semicolon is not recommended.

The "Cut" Operator (!)

- Automatic backtracking is one of the most characteristic features of Prolog.
  - Lead to inefficiency: explore possibilities that lead nowhere.
- The cut predicate tells the Prolog system to forget about some of the backtrack points.
  - Discards all backtrack points that have been recorded since execution entered the current clause.

The "Cut" Operator (!)

- After executing a cut:
  - It is no longer possible to try other clauses as alternatives to the current clause.
  - It is no longer possible to try alternative solutions to subgoals preceding the cut in the current clause.
- Reduces the search space.
  - "do not go to" (alternatives that we know are bound to fail).

```
writename(1) :- !, write('One').
```

- Confirms the choice of a rule.

```
max(X,Y,Y) :- X>=Y, !.
max(X,Y,X).
```


The "Cut" Operator (!)

- Simulates an "else" statement when we are testing cases.
  - Mutually exclusive conclusions .

```
f(X,0) :- X=0, !.
f(X,1) :- X>0, !.
f(X,undefined).
```

- Example: Given the knowledge base

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

What is the result of executing the query `?- f(a).` ?

Negative Goals ("not)

- The special predicate `\+` means "not" or "cannot prove".
- If g is any goal, then `\+ g` succeeds if g fails, and fails if g succeeds.

```
?- \+ likes(adan,apple).
```

- The predicate `\+` can appear only in a query or on the right-hand side of a rule.
  - It cannot appear in a fact or in the head of a rule.

```
\+ likes(adan,apple).
```


Negation as Failure

- The behaviour of `\+` is called *negation as failure.*
  - In Prolog, you cannot state a negative fact . All you can do is conclude a negative statement if you cannot conclude the corresponding positive statement.
    - What is the definition a a person who is not a parent?

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


Closed-World Assumption

- What happens if you ask about people who are not in the knowledge base?
- The Prolog systems assumes that its knowledge base is complete (this is called the closed-world assumption).
  - Something that cannot be proved to be true is assumed to be false.

Predicate "fail"

- Predicate *fail* is a special symbol that will immediately fail when Prolog encounters it as a goal.
- *Cut-fail* combination is used to say that something is not true.

```
likes(mary,X) :- snake(X), !, fail
likes(mary,X) :- animal(X).
```


Debugger / Tracer

- The debugger allows you to trace exactly what is happening as Prolog executes a query.

```
?- trace.
yes
```

  - *Return*: computation is shown step by step.
  - *s* (for "skip"): the debugger will skip to the end of the current query (useful if the current query has a lot of subgoals which you do not want to see).
  - *a* (for "abort"): the computation will stop.

Arithmetic

- Prolog supports both integers and floating-point numbers and interconvert them as needed.
- Operator "**is**": takes an arithmetic expression on its right, evaluates it, and unifies the result with its argument on the left.

```
?- Y is 2+2.          ?- 5 is 3+3.
Y = 4
yes                    no
?- Z is 4.5+(3.9/2.1).
Z = 6.3571428
yes
```

Arithmetic

- The precedence of operators is about the same as in other programming languages.
- Prolog is not an equation solver.
  - Prolog does not solve for unknowns on the right hand side of is:

```
?- 5 is 2 + What.
instantiation error.
```


Constructing Expressions

- Prolog vs. other programming languages
  - Other programming languages evaluate arithmetic expressions wherever they occur.
  - Prolog evaluates arithmetic expressions only in specific places.
    - 2+2 evaluates to 4 only when it is an argument of the predicates of the following table; the rest of the time it is just a data structure consisting of 2, +, and 2.

Built-in Predicates that Evaluate Expressions

`R is Expr`	Evaluates `Expr` and unifies result with `R`
`Expr1 =:= Expr2`	Succeeds if results of both expressions are equal.
`Expr1 =\= Expr2`	Succeeds if results of the expressions are not equal.
`Expr1 > Expr2`	Succeeds if `Expr1 > Expr2`
`Expr1 < Expr2`	Succeeds if `Expr1 < Expr2`
`Expr1 >= Expr2`	Succeeds if `Expr1 >= Expr2`
`Expr1 =< Expr2`	Succeeds if `Expr1 =< Expr2`

Constructing Expressions

- There is a clear difference between:
  - `is`, which takes an expression (on the right), evaluates it, and unifies the result with its argument on the left.
  - `=:=`, which evaluates two expressions and compares the results.
  - `=`, which unifies two terms (which need not be expressions and, if expressions, will not be evaluated).

Constructing Expressions: examples

```
?- What is 2+3.
What = 5        % Evaluates 2+3, unify result with What

?- 4+1 =:= 2+3.
yes             % Evaluates 4+1 and 2+3, compare results

?- What = 2+3
What = 2+3      % Unify What with the expression 2+3
```


Topics

- Examples
  - Execution trace
  - Controlling backtracking
- Lists

Lists

- One of the most important Prolog data structures.
- A list is an ordered sequence of zero or more terms written between square brackets and separated by commas.

```
[alpha,beta,gamma,delta]
[1,2,3,go]
[(2+2),in(austin,texas),-4.356,X]
[[a,list,within],a,list]
```


Lists

- The elements of a list can be Prolog terms of any kind, including other lists.
  - The element `[a]` is not equivalent to the atom a.
- The empty list is written `[]`.
- List can be constructed or decomposed through unification.
  - An entire list can match a single variable

Unify	With	Result
`[a,b,c]`	`X`	`X = [a,b,c]`

Lists

- Corresponding elements of two lists can be unified one by one.

Unify	With	Result
`[X,Y,Z]`	`[a,b,c]`	`X=a, Y=b, Z=c`
`[X,b,Z]`	`[a,Y,c]`	`X=a, Y=b, Z=c`

- This also applies to lists or structures embedded within lists.

Unify	With	Result
`[[a,b],c]`	`[X,Y]`	`X=[a,b], Y=c`
`[a(b),c(X)]`	`[Z,c(a)]`	`X=a, Z=a(b)`

Lists: head & tail

- Any list can be divided into head and tail by the symbol `|`
  - The head of a list is the first element
  - The tail is a list of the remaining elements (and can be empty).
  - *The tail of a list is always a list; the head of a list is an element.*

Lists: examples

- Every nonempty list has a head and a tail

```
[a|[b,c,d]] = [a,b,c,d]
[a|[]] = [a]
```

- The term `[X|Y]` unifies with any nonempty list, instantiating `X` to the head and `Y` to the tail

Unify	With	Result
`[X|Y]`	`[a,b,c,d]`	`X=a, Y=[b,c,d]`
`[X|Y]`	`[a]`	`X=a, Y=[]`

Lists: examples

Unify	With	Result
`[X,Y|Z]`	`[a,b,c]`	`X=a, Y=b, Z=[c]`
`[X,Y|Z]`	`[a,b,c,d]`	`X=a, Y=b, Z=[c,d]`
`[X,Y,Z|A]`	`[a,b,c]`	`X=a, Y=b, Z=c, A=[]`
`[X,Y,Z|A]`	`[a,b]`	`fails`
`[X,Y,a]`	`[Z,b,Z]`	`X=Z=a, Y=b`
`[X,Y|Z]`	`[a|W]`	`X=a, W=[Y|Z]`

Lists: internal representation

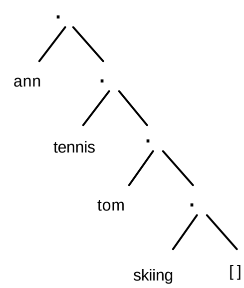
- The previous representation is only the external appearance of lists.
- All structured objects in Prolog are trees.
  - The head and the tail are combined into a structure by a special functor ‘.’

```
.(Head, Tail)
```

- Example:

```
[ann,tennis,tom,skiing]
.(ann,.(tennis,.(tom,.(skiing,[]))))
```


Lists: internal representation



- Both notations can be used.
- The square bracket notation is normally preferred.
- Internally, they are represented as binary trees.

Lists: recursion

- To fully exploit the power of lists:
  - A way to work with lists elements without specifying their positions in advanced.
- A repetitive procedure that will work their way along a list, searching for a particular element or performing some operation on every element encountered.
  - Repetition is expressed in Prolog by using *recursion*.

Lists: recursion

- In order to solve a problem, perform some action and then solve a similar problem of the same type using the same procedure.
- The process terminates when the problem becomes so small that the procedure can solve it in one step without calling itself again.
- Some common operations on lists: membership, concatenation, adding an item to a list, etc

Lists: membership

- A predicate `member(X,L)` succeeds if `X` is an element of the list `L`.
- Examples:

```
?- member(b,[a,b,c]).
   yes
?- member(b,[a,[b,c]]).
   no
?- member([b,c],[a,[b,c]]).
   yes
```


Lists: membership

- What can we say about the membership relation?
- In general, this relationship can be based on the following observation:

  X is a member of L if either

  (1) X is the head of L, or

  (2) X is a member of the tail of L.

Lists: membership

- Identify two special case that are not repetitive
  - If `L` is empty, fail with no further action (nothing is a member of the empty list).
  - If `X` is the first element of `L`, succeed with no further action (the element was found).
- To solve the first special case: make sure in all clauses that the second argument is something that will not unify with an empty list.

Lists: membership

- The second argument should be a list that has both a head and a tail.
  - The second special case (a simple clause):

```
member(X,[X|Tail]).
```

- The recursive part ("X is a member of L if X is a member of the tail of L") can be expressed as:

```
member(X,[Head|Tail]):-
        member(X,Tail).
```


Lists: concatenation

- The predicate concatenate or append combines to lists into a single list.
- Examples:

```
?- concatenate([a,b,c],[d,e,f],X).
   X = [a,b,c,d,e,f].
```

- Can we use '|'? `[[a,b,c]|[d,e,f]]` is equivalent to `[[a,b,c],d,e,f]`.
- Strategy: work through the first list element by element, adding elements one by one to the second list.

Lists: concatenation

- The definition `concatenation(L1,L2,L3)` will have again two cases, depending on the first argument, `L1`:

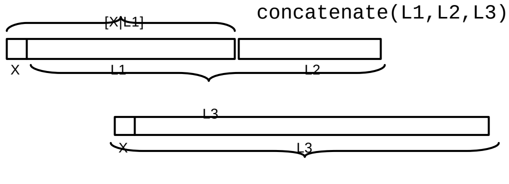
  (1) Since the first list will be shortened, it will eventually become empty. So, if the first argument is an empty list then the second and the third arguments must be the same list.

```
concatenate([],L,L).
```


Lists: concatenation

(2) If the first argument is a non-empty list then it has a head and a tail. The new list have the same head and the concatenation of the tail with the second list.

```
concatenate([X|L1],L2,[X|L3]):-
        concatenate(L1,L2,L3).
```



Lists: concatenation

- Examples:

```
?- concatenate([a,[b,c],d],[a,[],b],X).
   X = [a,[b,c],d,a,[],b]
?- concatenate([a,b,c],X,[a,b,c,d,e,f]).
   X = [d,e,f]
?- concatenate(X,[d,e,f],[a,b,c,d,e,f]).
   X = [a,b,c]
?- concatenate(X,Y,[a,b,c,d]).
   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 = []
```

Lists: adding an item

- To add an item to a list, it is easier to put the new item in front of the list so that it becomes the new head.

```
add(X,L,[X|L]).
```

- Examples:

```
?- add(0,[1,2,3],L).
   L = [0,1,2,3]
?- add(X,[b,c,d],[a,b,c,d]).
   L = a
```


Lists: deleting an item

- The delete operation `delete(X,L,L1)` deletes an item X from a list L, where L1 is equal to the list L with the item X removed.
  - (1) If X is the head of the list then the result after the deletion is the tail of the list.

```
delete(X,[X|Tail],Tail).
```

  - (2) If X is in the tail then it is deleted from there.

```
delete(X,[Y|Tail],[Y|Tail1]):-
                     delete(X,Tail,Tail1).
```


Lists: deleting an item

- `Delete`, like `member`, is also non-deterministic in nature.
  - If there are several occurrences of X in the list then delete will be able to delete anyone of them by backtracking.
  - Each alternative execution will only delete one occurrence of X, leaving the others untouched.

```
?- delete(a,[a,b,a,a],L).
L = [b,a,a];
L = [a,b,a];
L = [a,b,a];
no
```


Lists: counting list elements

- This is the recursive algorithm to count elements of a list:
  - If the list is empty, it has 0 elements.

```
list_length([],0).
```

  - Otherwise, skip over the first element, count the number of elements remaining and add 1.

```
list_length([_|Tail],K) :-
                list_length(Tail,J).
                K is J+1.
```


Lists: reversing a list

- Recursive algorithm for reversing the order of elements in a list:
  - (1) Split the original list into a head and tail.
  - (2) Recursively reverse the tail of the original list.
  - (3) Make a list whose only element is the head of the original list.
  - (4) Concatenate the reversed tail of the original list with the list created in step 3.

Lists: reversing a list

- The list gets shorter every time, the limiting case is an empty list, which Prolog must return unchanged.

```
reverse([],[]).
reverse([Head|Tail],Result) :-
   reverse(Tail,ReverseTail),
   concatenate(ReverseTail,[Head],Result).
```

- Example:

```
?- reverse([a,b,c,d],X).
   X = [d,c,b,a]
```

Family Facts

```
parent(pam,tom).        female(pam).
parent(tom,bob).        male(tom).
parent(tom,liz).        male(bob).
parent(bob,ann).        female(liz).
parent(bob,pat).        female(ann).
parent(pat,jim).        female(pat).
                        male(jim).
```


Controlling backtracking: example

- Consider the double step function. The relation between X and Y can be specified by three rules:
  - Rule 1: if X < 3 then Y = 0
  - Rule 2: If 3 ≤ X and X < 6 then Y = 2
  - Rule 3: if 6 ≤ X then Y = 4

Controlling backtracking: experiment 1

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

- Question: `?- f(1,Y), 2 < Y.`
- The first goal `f(1,Y), Y` becomes instantiated to `0`.
- The second goal becomes `2 < 0` which fails, and so does the whole goal list.
- Before admitting that the goal list is not satisfiable, Prolog tries, through backtracking, two useless alternatives.
- The three rules about the `f` relation are mutually exclusive so that one of them at most will succeed.

Controlling backtracking: experiment 2

```
f(X,0) :- X < 3, !.
f(X,2) :- X < 6, !.
f(X,4).
```

- Equivalent to these three rules:

  if X <3 then Y =0,

  otherwise if X < 6 then Y = 2,

  otherwise Y = 4

Examples using cut

- Computing maximum

  ```
  max(X,Y,X) :- X >= Y, !.
  max(X,Y,Y).
  ```
- Single-solution membership

  ```
  member(X,[X|L]) :- !.
  member(X,[Y|L) :- member(X,L).
  ```
- Classification into categories

```
class(X, fighter) :- beat(X,_),beat(_,X),!.
class(X,winner) :- beat(X,_), !.
class(X, sportsman) :- beat(_,X).
```


Input and Output

- The build-in predicate **`read`** is used for reading terms from the current input.
- The goal **`read(X)`** will cause the next term, `T`, to be read, and this term will be matched with `X`.
  - If `X` is a variable then `X` will be instantiated to `T`.
  - If matching does not succeed the the goal fails.

Input and Output

- This predicate is deterministic, so in the case of failure there will be no backtracking to input another term.

- The build-in predicate **`write`** is used for writing terms to the current output.
  - This predicate 'knows' to display any term no matter how complicated it may be.

Constructing and Decomposing Atoms

- An atom can be converted to a sequence of characters using the build-in predicate **`name`**.
  - This predicate relates atoms and their ASCII codes.
  - `name(zx232,[122,120,50,51,50]).`
- Two typical uses:
  1. Given an atom, break it down into single characters.
  2. Given a list of characters, combine them into an atom

Testing the Type of Terms

- Sometimes it is useful to know what is the type of some value.
- Example: if we want to add the values of two variables `X` and `Y` by: **`Z is X + Y.`**
  - Before this goal is executed, `X` and `Y` have to be instantiated to integers.
- The build-in predicate `integer(X)` is true if `X` is an integer or if it is a variable whose value is an integer.
  - `X` must 'currently stand for' an integer.

Testing the Type of Terms

- **`var(X)`** succeeds if `X` is currently an uninstantiated variable.
- **`nonvar(X)`** succeeds if `X` is a term other than a variable, or `X` is an already instantiated variable.
- **`atom(X)`** is true if `X` currently stands for an atom.
- **`atomic(X)`** is true if `X` currently stands for an integer or an atom.

Testing the Type of Terms

- **`compound(X)`** succeeds if `X` is a compound term (a structure, including lists but not []).
- **`number(X)`** succeeds if `X` is a number (integer of floating-point).
- **`float(X)`** succeeds if `X` is a floating-point number.

Constructing and Decomposing Terms

- There are three build-in predicates for decomposing terms and constructing new terms.
  - **`Term=..L`** is true if `L` is a list that contains the principal functor of `Term`, followed by its arguments.
  - **`functor(Term,F,N)`** is true if `F` is the principal functor of `Term` and `N` is the arity of F.
  - **`arg(N,Term,A)`** is true if `A` is the `Nth` argument in `Term`, assuming that arguments are numbered from left to right starting with 1.

Finding all Solutions to a Query

- Prolog can generate, by backtracking, all the objects, one by one, that satisfy some goal.
  - Each time a new solution is generated, the previous one disappears and is not accessible any more.
  - Sometime we would prefer to have all generated objects available together.

Finding all Solutions to a Query

- **`findall(T,G,L)`** find each solution to `G`; instantiates variables to `T` to the values that they have in that solution; and adds that instantiation of `T` to `L`.
- **`bagof(T,G,L)`** like `findall` except for its treatment of the free variables of `G` (those that do not occur in `T`).

Finding all Solutions to a Query

  - Whereas `findall` would try all possible values of all variables, `bagof` will pick the first set for the free variables that succeeds, and use only that set of values when finding the solution in `L`.
  - If you ask for an alternative solution to `bagof`, you will get the results of trying another set of values for the free variables.
- **`setof(T,G,L)`** like `bagof` but the elements of `L` are sorted into alphabetical order and duplicates are removed.