Logic Programming & Prolog

- Introduction to predicate calculus
- Logic programming
- Prolog
- Features of logic programming
Main Features of Logic Programming

• An example:

```
ancestor(X, Y) :- parent-child(X, Y).
ancestor(X, Y) :- parent-child(X, Z), ancestor(Z, Y).
```

• A non-sequential programming language

• Declarative - program defines what the problem is, rather than how to solve it

• Symbolic and recursive
The simplicity of Prolog relative to other languages

- Very little syntax
- Theoretically no data types (?)
- Very clear semantics - defined mathematically

Why use Prolog?

- Declarative nature allows programmer to concentrate on the nature of the problem, rather than the method of solution.
- Some pragmatics are required however to produce advanced problem solutions!!
References


• Bratko I. (2011), Prolog: Programming for Artificial Intelligence, Addison-Wesley, Reading, MA.

1. Introduction to Predicate Calculus

- The propositional calculus
- The predicate calculus
- Using inference rules to produce predicate calculus expressions (omitted)
The Propositional Calculus: Syntax 1/3

• Propositional symbols: \( P, Q, R, S, T, \ldots \) denote *propositions*, or statements about the world that maybe either true or false, such as “Xindong is female” or “Australia is a country”.

• Truth symbols: \textit{true}, \textit{false}

• Connectives: \( \land \) (and), \( \lor \) (or), \( \neg \) (not), \( \Rightarrow \), \( = \)

• Propositional calculus *sentences*:
  - Every propositional symbol and truth symbol is a sentence. E.g., \textit{true}, and \( Q \).
  - The *negation* of a sentence is a sentence. E.g., \( \neg P \).
  - Any 2 sentences connected by one of \( \{ \land, \lor, \Rightarrow, = \} \) are a sentence. E.g., \( P \lor \neg Q \Rightarrow R \).

Legal sentences are called *well-formed formulas* or *WFFs*. 
The Propositional Calculus: Syntax (2)

• In expressions of the form $P \land Q$, $P$ and $Q$ are called *conjuncts*.

• In $P \lor Q$, $P$ and $Q$ are called *disjuncts*.

• In $P \Rightarrow Q$, $P$ is the *premise* or *antecedent*, and $Q$ is the *conclusion* or *consequent*.

• The symbols ( ) and [ ] are used to group symbols into subexpressions and so control their order of evaluation and meaning. $(P \lor \neg Q) = R$ and $P \lor (\neg Q = R)$ are different.
The Propositional Calculus: Semantics

- The *Semantics* or meaning of sentences:
  
  - A proposition of symbol corresponds to a statement about the world. It can be either true or false, given some state of the world.
  - The truth value assignment to propositional sentences is called an *interpretation*, which is a mapping from the propositional symbols into the set \{T, F\}. Note that \{T, F\} and \{true, false\} have been intentionally used for different purposes.
  - Each possible mapping corresponds to a possible world of interpretation. E.g., if \(P\) denotes “it is raining” and \(Q\) denotes “I am at work”, then the set of propositions \{\(P, Q\)\} has 4 different functional mappings into the truth values \{T, F\}, which correspond to 4 different possible worlds.
The Propositional Calculus: Semantics (2)

• Propositional calculus semantics:
  – An interpretation of a set of propositions is the assignment of a truth value, either \( T \) or \( F \), to each propositional symbol.
  – The symbol \( \text{true} \) is always assigned \( T \), and \( \text{false} \) always \( F \).
  – The truth assignment of \( \text{negation} \), \( \neg P \), is \( F \) if the assignment to \( P \) is \( T \), and \( T \) if the assignment of \( P \) is \( F \).
  – The truth assignment of \( \text{conjunction} \), \( \land \), is \( T \) only when both conjuncts have truth value \( T \); otherwise it is \( F \).
  – The truth assignment of \( \text{disjunction} \), \( \lor \), is \( F \) only when both disjuncts have truth value \( F \); otherwise it is \( T \).
  – The truth assignment of \( \text{implication} \), \( \Rightarrow \), is \( F \) only when the premise is \( T \) and the consequent is \( F \); otherwise it is \( T \).
  – The truth assignment of \( \text{equivalence} \), \( \equiv \), is \( T \) when both expressions have the same truth assignment for all possible interpretations; otherwise it is \( F \).

• The truth assignments of compound propositions are often described in \( \text{truth tables} \).
The Propositional Calculus: Semantics (3)

- A *truth table* lists all possible truth value assignments to the atomic propositions of an expression and gives the truth value of the expression for each assignment.

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>¬P</th>
<th>¬P ∨ Q</th>
<th>P ⇒ Q</th>
<th>(¬P ∨ Q) = (P ⇒ Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

- An example:

- By demonstrating that they have identical truth tables, we prove that 2 (or more) expressions are equivalent.

\[ \neg(\neg P) = P \]
\[ (P \lor Q) = (\neg P \Rightarrow Q) \]
\[ \neg(P \lor Q) = (\neg P \land \neg Q) \]
\[ \neg(P \land Q) = (\neg P \lor \neg Q) \]
The Predicates Calculus

• In propositional calculus, each atomic symbol (such as $P$ and $Q$) denotes a proposition of some complexity. There is no way to access the components of an individual assertion. $P = \text{“Xindong’s car has 5 doors”}$.

• In predicate calculus, we can put it as $\text{car_door(xindong, 5)}$ and $\text{car_door(X, 5)}$.

• To be continued in the context of Prolog ...
2. (Turbo) Prolog: Notational Conventions (1/5)

- Terms: All data objects are called terms.
- Atomic terms: (this is a misnomer as in logic, predicates are called atoms and atoms are called constants. However, we’ll stick to the Prolog convention.)
  - characters, such as ‘a’ in Turbo Prolog.
  - integers, e.g., anything between -32768 and 32767.
  - reals, such as 86.92, -9111.929437, -79.84e-21.
  - strings enclosed in ”double quotes”.
  - symbols in one of the following 2 forms: (1) alphanumerics and _, starting with a lower case alphabetic, and (2) enclosed in ”double quotes” as strings.
  - files.
Notational Conventions of (Turbo) Prolog (2)

• Function terms:
  – Functions have the form
    \( <\text{functor}>\{((<\text{term}>\{,<\text{term}>\})\})^1_0 \).
  – Functors are symbols and start with a lower case alphabetical.
  – Examples:
    hello
    grade_attained_in(ai,pass)
  – The number of arguments is the arity of the function. When referring to a predicate, it is written with its arity in the format \( <\text{functor}>/<\text{arity}> \). This is also true for atoms, whose arity is 0.
Notational Conventions of (Turbo) Prolog (3)

- Composite terms: e.g.,

  owns(xindong,
    book("Knowledge Acquisition from Databases",
         "Ablex, USA", 1995)).

  - The definition is recursive.
  - The view of functions as trees.
  - Operators:
    * Some functors are used in infix notation, e.g. 5+4.
    * Operators do not cause the associated function to be carried out.
Notational Conventions of (Turbo) Prolog (4)

• Variables:
  – Uppercase or _ for start of variables
  – Examples:
    Who
    What
    _someone
  – Variables in Prolog are rather different to those in most other languages. Further discussion and use is deferred until later.
Notational Conventions of (Turbo) Prolog (5)

- Predicates: a more general term of functors.
  - A predicate has the form
    \[ <\text{predicate symbol}>\{(<\text{term}>\{,,<\text{term}>\})\}_0^1. \]
  - The predicate symbol starts with a lower case alphabetic and the \(<\text{term}>s\) can be variables.
  - A degenerate form of predicate is a proposition, which has no arguments, i.e., \(<\text{predicate symbol}>\).
  - The number of arguments is the arity of the predicate. When referring to a predicate, it is written with its arity in the format \(<\text{predicate symbol}>/<\text{arity}>\). This is also true for propositions, whose arity is 0.
Prolog Programs – Clauses (1/5)

- Programs are made up of Horn clauses, which in turn are made up of predicates. The closest imperative structure is a ’Boolean function’.
- The heading of a clause is a single predicate (also known as the consequent).
- The body of a clause is a sequence of 0 or more predicates, separated by commas (also known as the antecedent). These are ’calls’, called queries in Prolog, to other clauses.
- If there are predicate(s) in the body, the body is separated from the head with :−.
Prolog Programs – Clauses (2)

- Clauses with 0 body predicates are facts and those with 1 or more body predicates are rules.
- Facts may be considered to be rules with the predicate true as the tail.
- Whenever a clause is used, new copies of its variables are created (so they are local), typically by adding a numeric suffix.
Prolog Programs – Clauses (3)

• Facts
  – Propositional facts: <proposition>.
    * Always end with . in a program.
    * Lower case alphabetic for first letter of predicate symbols.
    * Example: xindong_is_a_lecturer.
  – Predicates with no variable arguments:
    * <predicate symbol>(<arguments>).
    * Always end with . in a program.
    * Examples:
      lecturer(xindong,ai).
      lecturer(tim,prerequisite_to(ai)).
Prolog Programs – Clauses (4)

- Propositional rules:
  - Always end with . in a program.
  - Head separated from body by :- (if)
  - Body predicates are separated by , (and)
  - Example:

    can_study_adv_ai:- has_studied_ai.
    can_study_adv_ai:-
      hod_says_so.
    will_study_adv_ai:-
      can_study_adv_ai,
      enrolled_in_adv_ai.
Prolog Programs – Clauses (5)

- Rules with no variables:
  - As for propositional rules, but with arguments
  - Example:

```prolog
can_study(xindong, adv_ai):-
    has_studied(xindong, prerequisite_to(adv_ai)).
can_study(xindong, adv_ai):-
    hod_permission_to_study(xindong, adv_ai).
will_study(xindong, adv_ai):-
    can_study(xindong, adv_ai),
    enrolled(xindong, adv_ai).
```
Queries (1/2)

Queries, in the form of a clause body, are entered at the `?-` prompt, and end with a stop (`.`).

- A single query succeeds if there is a clause, whose head unifies with the query, which succeeds. In the absence of variables unifies simply means that they must match exactly (The concept unification will be generalized later).

- A conjunctive query succeeds if each element succeeds (an empty conjunction succeeds immediately). The members of a conjunctive query are evaluated sequentially, and evaluation is stopped as soon as a failed member is encountered.
Queries (2)

- A clause succeeds if its body succeeds. Note that bodies are the same as conjunctive queries.
- Queries will eventually succeed or fail and the interpreter will print yes or no (or true or false) appropriately.
- If no head unifies with a query, then the query fails.
Query Examples (1/2)

Queries on propositional facts:

```
xindong_is_a_lecturer.

?-xindong_is_a_lecturer.
yes
?-alan_is_a_lecturer.
no
```

Query on propositional rules:

```
has_studied_ai.
has_brains_to_pass_adv_ai.
enrolled_in_adv_ai.
can_study_adv_ai:-
    has_studied_ai.
will_study_adv_ai:-
    can_study_adv_ai,
enrolled_in_adv_ai.

?-will_study_adv_ai.
yes
```
Query Examples (2)

Conjunctive queries:

```prolog
has_studied_ai.
has_brains_to_pass_adv_ai.
can_study_adv_ai:~
  has_studied_ai.
will_study_adv_ai:~
  can_study_adv_ai,
  enrolled_in_adv_ai.
```

```
?-will_study_adv_ai, has_brains_to_pass_adv_ai.
no
```

Query on rules with no variables:

```prolog
has_studied(xindong,prerequisite_to(adv_ai)).
can_study(xindong,adv_ai):~
  has_studied(xindong,prerequisite_to(adv_ai)).
```

```
?-can_study(xindong,adv_ai).
yes
```
Backtracking (1/2)

• There may be more than one version of a clause. The first version (in source code order) is used when the clause is queried. If the first version fails, then the second is tried, and so on.

• The query fails if none of the versions succeed. This may lead to alternatives further back being tried, and so on. This process is called backtracking.

• Example: Query with backtracking on failure.

```
hod_says_so.
can_study_adv_ai:-
    has_studied_pascal.
can_study_adv_ai:-
    hod_says_so.

?-can_study_adv_ai
yes
```
Backtracking (2)

As well as finding a solution, backtracking can also find alternative solutions. After a solution is given, the user can ask for alternatives to be found. The next alternative of the last call that succeeded, is used. If it does not have any alternatives, then calls further back are tried successively. (A ; is used to get alternative answers in SICStus Prolog.)

Example with SICStus:

```prolog
has_studied_pascal.
hod_says_so.
has_studied_pascal.
enrolled_in_adv_ai.
has_brains_to_pass_adv_ai.
can_study_adv_ai:-
    has_studied_pascal.
can_study_adv_ai:-
    hod_says_so.
will_study_adv_ai:-
    can_study_adv_ai,
enrolled_in_adv_ai.
```

```
?- will_study_adv_ai, has_brains_to_pass_adv_ai.
yes ;
yes ;
yes
```
Variables (1/2)

The only way to introduce variables is in a formal or actual argument list. Variables are local to their clause, and each time a clause is used a new set of variables are allocated. The new variables are typically appended with a unique number to make them distinct and local.

Clauses with variables:

- Variables in clauses are universally quantified
- Example

```prolog
has_studied(Everyone, cs21).

can_study(Anyone, Anything):-
    has_studied(Anyone, prerequisite_to(Anything)).

can_study(Anyone, Anything):-
    hod_permission_to_study(Anyone, Anything).
```
Variables (2): Unification

Variables can be in one of two states - bound (have value) or unbound (no value). Variables are initially unbound. Bound variables cannot have their values reset. However bound variables can become unbound (in the case of backtracking). The binding of variables occurs in a process called **unification**.

Two values/variables unify iff

1. They are identical, or
2. They are both "functions" with the same functor, and their arguments pairwise unify, or
3. One is a variable, in which case the the other value/variable is bound to the first.

When two variables unify and one is bound to the other, the result is similar to the passing of reference parameters, and any change made to one is made to the other. The variables are said to share. When a clause is queried, the actual and formal arguments must pairwise unify.
Query with Unification: Examples

lecturer(xindong,ai).
---------------------------------
?-lecturer(xindong,ai).
yes
?-lecturer(heinz,ai).
no
?-lecturer(xindong,networks).
no

As well as answering yes or no, the bindings of any variables in the original query are printed by the interpreter. Variables in queries are existentially quantified.

lecturer(xindong,ai).
---------------------------------
?-lecturer(xindong,What).
yes
What = ai
?-lecturer(Who,ai).
yes
Who = xindong
?-lecturer(Who,What).
yes
Who = xindong
What = ai
?-lecturer(Same,Same).
no
Query with Variables and Backtracking on Failure

When another version of a clause is used in backtracking, any variable bindings done in the use of the previous version are undone.

```prolog
lecturer(tim,ai).
lecturer(xindong,ai).
rides_motorcycle(xindong).

?-lecturer(Anyone,ai),rides_motorcycle(Anyone).
yes
Anyone = xindong

has_studied(joe,prerequisite_to(adv_ai)).
hod_permission_to_study(jane,adv_ai).
can_study(Anyone,Anything):-
    has_studied(Anyone,prerequisite_to(Anything)).
can_study(Anyone,Anything):-
    hod_permission_to_study(Anyone,Anything).

?-can_study(jane,Anything).
yes
Anything = adv_ai
```
Query with Variables and Backtracking on Success

lecturer(tim,ai).
lecturer(john,pascal).
lecturer(xindong,ai).
lecturer(john,programming).
lecturer(andy,networks).
lecturer(peter,graphics).
lecturer(xindong,projects).
lecturer(tim,ppl).
lecturer(john,graphics).

?-lecturer(Anyone,Subject),lecturer(Anyone,Another_subject).

yes
Anyone = tim
Subject = ai
Another_subject = ai

yes
Anyone = tim
Subject = ai
Another_subject = ppl

yes
...
Comments (inside programs)

Two forms:

/* ---- This is a comment. */

% ----This is a comment.
Exercises

Write a program that stores information about your family, and will answer queries about relationships. Information about individual people must be stored as facts containing the persons name, sex and parents’ names, e.g. person(charles,male,xindong,yan).

Here are the queries you should be able to answer ...

• ?-sibling(Person,Sibling). Hint: Siblings have the same parents.
• ?-brother(Person,Brother). Hint: A brother is a male sibling.
• ?-sister(Person,Sister). Hint: A sister is a female sibling.
• ?-father(Person,Father).
• ?-mother(Person,Mother).
• ?-parent(Person,Parent). Hint: Either a mother or father
• ?-offspring(Person,Offspring). Hint: The inverse of parent.
• ?-son(Person,Son). Hint: A son is a male offspring.
• ?-daughter(Person,Daughter). Hint: A daughter is a female offspring.
• ?-aunt(Person,Aunt). Hint: An aunt is a parent’s sister.
• ?-uncle(Person,Uncle). Hint: An uncle is a parent’s brother.
• ?-nephew(Person,Nephew). Hint: A nephew is a sibling’s son.
• ?-niece(Person,Niece). Hint: A niece is a sibling’s daughter.
• ?-descendant(Person,Descendant). Hint: A descendant is an offspring or a descendant’s offspring.
• ?-ancestor(Person,Ancestor). Hint: The inverse of descendant.
• ?-relation(Person,Relation).
More about Terms

• The anonymous variable: The single underscore variable (_). It is used when the value of the variable is not cared about.

```prolog
lecturer(tim,ai).
lecturer(john,pascal).
lecturer(xindong,ai).
lecturer(john,programming).
lecturer(damien,networks).
lecturer(peter,graphics).
lecturer(xindong,projects).
lecturer(tim,ppl).
lecturer(john,graphics).
```

?‐lecturer(Anyone,_).
  yes
  Anyone = tim  ;
  yes
  Anyone = john  ;
  yes ...

Classifying Terms

It is often useful to determine the nature of a term, especially of an instantiated variable.

\[
\text{var}(X) \ (\text{free}(X)) \text{ checks for unbound variables.}
\]
\[
\text{nonvar}(X) \ (\text{bound}(X)) \text{ checks for non-unbound variables}
\]

SICStus Prolog only:

\[
\text{atom}(X) \text{ checks for atoms}
\]
\[
\text{integer}(X) \text{ checks for integers}
\]
\[
\text{atomic}(X) \text{ checks for integers or atoms.}
\]
Unification and Instantiation

• Unification is the operation that compares two expressions, and, if possible, instantiates (or binds) variables in the expressions so that the two expressions become identical. Before instantiation a variable is unbound, but once instantiated the variable is completely replaced by the term it has been instantiated to.

• Unification is used to compare query predicates with clause heads. In unifications, the instantiation of variables is as general as possible, to create a common instance of the two expressions.

For example, \( p(X, a, Z) \) and \( p(A, B, B) \). The most common instance is \( p(X, a, a) \), but \( p(a, a, a) \) is also a common instance.
The Rules for Unification (1)

- Functions/predicates unify if their functors/predicate symbols and arities are identical, and their arguments unify.

- Variables unify with any term not containing that variable. The variable is instantiated to that term. (The check for the variable in the term is called the occur check, and is not implemented by most Prologs for efficiency reasons. This causes problems in examples such as $p(X, X)$ with $p(Y, f(Y))$.) The = predicate determines if two expressions are unifiable, and <> the converse.

• Examples

  ?- p(a, X, f(Y)) = p(Z, Z, f(b)).
  yes
  X = a
  Y = b
  Z = a
The Rules for Unification (2)

- Note that the following is not allowed, and will not work. The reason is that it is second order logic.

```prolog
?-lecturer(xindong,ai) = Job(xindong,ai).
yes
Job = lecturer
```

- Check for identical and non-identical expressions.
- Distinct variables are not identical, but unified ones are.
- Compare these against = and <>.
Debugging

\texttt{trace(on)} (\texttt{trace} in Turbo Prolog) turns on tracing
\texttt{trace(off)} turns off tracing
\texttt{trace \{p1,p2,...\}} as the first line of your program
with Turbo Prolog
**Arithmetic (1/3)**

- Evaluation is caused by the $=$ predicate.
- The RHS of $=$ may contain any of the arithmetic operators $+$, $-$, $\ast$, $/$, mod.
- The RHS is evaluated, and the result unified with the LHS.
- The RHS must be evaluable, i.e., no unbound variables.
- Example:
  ?- X=2+3+4.
  yes
  X = 9
- The arithmetic predicates $=$, $<$, $>$, $\geq$, $\leq$ and $<>$ can be used to compare evaluable arithmetic expressions.
Arithmetic (2): Examples

- To find the greatest common divisor of X and Y. (Hint: If X and Y are equal, then that is the GCD, else find the GCD of the smaller of X and Y, and the absolute of their difference).

```prolog
gcd(X,X,X).
gcd(X,Y,GCD):-
    X < Y,    % X is less than Y
    Difference = Y - X,
    gcd(X,Difference,GCD).
gcd(X,Y,GCD):-
    gcd(Y,X,GCD).
```
Arithmetic (3): Exercises

Write clauses for sum and prod. sum takes three arguments, the first two being operands, the last being the sum of the first two.

- If all arguments are bound then sum must check that the first two add to the third.
- If the first two arguments are bound and the last a variable, sum must add the first two and bind the result to the third.
- If either of the first two are variables, and the other two are bound, then sum must perform the subtraction and bind the variable appropriately.
- Otherwise sum must fail.
- prod should perform similarly for multiplication and division.
Recursion

- Direct recursion: The same term appears in the definition of the term itself.
- Indirect recursion: A is defined on B, and B uses A in its definition.
- An example for direct recursion:

  \[
  \text{factorial}(1,1).
  \text{factorial}(N,X) :- \]
  \[
  N \text{>1, } N1=N-1,
  \text{factorial}(N1,X1), X=N \times X1.
  \]

- Two essential parts for direct recursion:
  - An exit part where the conditions are met.
  - A recursion body where the conditions/counters are changed.
Lists (1/7)

• A list consists of 2 parts: the first part is the head and the second part is the tail of the list. \([<\text{head}>,<\text{more head}>[|<\text{tail list}>]]\) - i.e. any number of head elements, optionally followed by a | and a tail list, all enclosed in \([]\).

• The head may be anything, the tail must be a list. This means that the elements of a list need not be homogeneous, and can be anything, even lists.

• The innermost function has an empty list as its tail. The empty list is written \([]\).

• To declare an integer list in Turbo Prolog:

```prolog
domains

integerlist = integer*
```

* indicates 0, 1 or more elements of the same type.
Lists (2): Examples

numbers([1,2,3,4]).
sentence(the,[cat,sat|[on,the,mat]]).

?-numbers([X|Y]).
yes
X = 1
Y = [2,3,4]

?-numbers([1,Y|Z]).
yes
Y = 2
Z = [3,4]

?-sentence(the,[Noun|Rest]).
yes
Noun = cat
Rest = [sat,on,the,mat]
Lists (3): member/2

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

The first clause can be rewritten into a more understandable format for beginners.

Three interpretations:

• Is \( E \) an element of \( L \)?
• What are the elements of \( L \)?
• What lists have \( E \) as an element (a dangerous interpretation with an infinite number of answers)?
Lists (4): append/3

append([],List,List).
append([Head|Tail],List,[Head|Tail_list]):-
    append(Tail,List,Tail_list).

This program can be rewritten into a more understandable format for beginners.

Five interpretations:

• What is L1 appended to L2?
• What must be appended to L1 to get L?
• What must be prepended to L2 to get L?
• What two lists append to form L?
• What two lists append to form another list (infinite number of answers)?
Lists (5): length/2

length([],0).
length([Head|Tail],Length):-
    length(Tail,Tail_length),
    Length is Tail_length + 1.

Three interpretations:

- What is the length of L?
- Does L have this length?
- What Ls have this length?
Lists (6): multiple uses of the same procedures

• Example 1: `append` can implement `member`

  \[
  \text{member}(	ext{Element}, \text{List}) : - \\
  \text{append}(_, [\text{Element} | _], \text{List}).
  \]

• Example 2: `append` can implement `sublist`

  \[
  \text{sublist}(\text{Sublist}, \text{List}) : - \\
  \text{append}(_, \text{Back}, \text{List}), \\
  \text{append}(\text{Sublist}, _, \text{Back}).
  \]
Lists (7): Exercises

• Write clauses for mega_append, which has a list of lists as its first argument and their concatenation (all appended) as the second argument.

• Write a program to solve the Towers of Hanoi problem. Your program should generate a list of functions of the form move(From_peg, To_peg), returned in the last argument of hanoi(From_peg, To_peg, Spare_peg, Number_of_disks, List_of_moves_to_solve).
fail

- **fail** always fails. It forces backtracking on the preceding predicate.
Negation by Failure (1/6)

• \( \bot \) (not in Turbo Prolog) tests if its argument fails as a query. (\(<>\) compares 2 atomic terms.)

• The basis of negation by failure is the closed world assumption (CWA). The CWA is non-monotonic, i.e. the addition of new axioms may reduce the amount of information derivable.
Negation by Failure (2): Examples

clever(X):-
   
   \+(stupid(X)).

?-clever(xindong).
yes

then add the fact stupid(xindong).

clever(X):-
   \+(stupid(X)).
stupid(xindong).

?-clever(xindong).
no

The fact clever(xindong) is no longer derivable.
Negation by Failure (3): Examples

Negation by failure is not as powerful as the CWA.

```
s(a).
q(b).
q(X):- r(X).
r(X):- q(X).
```

The set of true things is \{s(a), q(b), r(b)\}. Thus under CWA \texttt{not(q(a))} is true. But negation by failure cannot find this as the search is infinite. Negation by failure requires finite failure.
Negation by Failure (4): Non-ground negated goals

\[ r(a). \\
q(b). \\
p(X):- \neg(r(X)). \]

\[ ?-q(X),p(X). \]
yes
\[ X = b. \]

\[ ?-p(X),q(X). \]
no

The problem in the second query is that \( X \) is unbound in \( \neg(r(X)) \).
Negation by Failure (5): Non-ground negated goals

- \texttt{not(not(p))} is not always equivalent to \texttt{p}!

\begin{verbatim}
\texttt{p(a).}
\texttt{-------------}
\texttt{?-p(X). yes X = a.}
\texttt{?-\+(\+(p(X))).yesX = X.}
\end{verbatim}

- The problem is that the negated goal is non-ground. Thus, for \texttt{\+} to work
  - Its argument must contain no variables. If they do, no binding will be returned even if the negated goal succeeds.
  - It must not get into an infinite loop (true for positive goals too of course).
Negation by Failure (6): Exercises

Write `difference_member(Elements, Allowed, Not_allowed)` which finds/checks for `Elements` that are members of the list `Allowed` but not members of the list `Not_allowed`.

Two examples:

```
difference([2,3,4],[1,2,4,3,5],[6,9,3,7]). false.
difference(X, [1,2,4,3,5], [6,9,3,7]). true.
X=[1,2,4,5].
```
The Cut (1/8)

The cut is a mechanism for indicating that certain predicates will not have (you do not want them to have) any other solutions.

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

This definition of `member` will, through backtracking, find all occurrences of the element in the list. If the list is a set and the first clause succeeds, there is no point in pursuing the second. A cut can be used to eliminate the excess backtracking.

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

This illustrates the first effect of the `cut`: any other alternative clauses should not be considered.
The Cut (2): union

union([],Set,Set).
union([Element|Rest_of_set1],Set2,Union):-
    member(Element,Set2),
    union(Rest_of_set1,Set2,Union).
union([Element|Rest_of_set1],Set2,
    [Element|Rest_of_union]):-
    union(Rest_of_set1,Set2,Rest_of_union).

This **union** has the problem that it will give unions with duplicate elements on backtracking.

Further, if member succeeds more than once (i.e., it is a general version, like the former above) then it will also give the correct union more than once. A cut after the member in the second clause would solve this.
The Cut (3): union

union([],Set,Set).
union([Element|Rest_of_set1],Set2,Union):-
    member(Element,Set2),
    !,
    union(Rest_of_set1,Set2,Union).
union([Element|Rest_of_set1],Set2,Set2,Rest_of_union):-
    union(Rest_of_set1,Set2,Rest_of_union).

This illustrates the first and second effects of the cut: (i) any other alternative clauses should not be considered; (ii) the solution obtained so far, for the tail predicates of this clause, is correct.
The Cut (4): A menu

menu:-
    write("Enter menu option : "),
    readln(Option),
    legal_option(Option),
    !,
    do_option(Option),
    menu.
do_option(foo):- !,
    write(foo), nl.
do_option(blah):- !,
    write(blah), nl.
do_option(Unknown)
    write(Unknown),
    write(" is an unknown option").
The Cut (5): cut implementing if-then-else

factorial(N,Answer):-
    N > 1,
    !,
    N1 = N - 1,
    factorial(N1,N1_answer),
    Answer = N * N1_answer,
    write(Answer, " is calculated for "),
    write(N),nl,
    nl.
factorial(N,1):-
    write("1 is calculated for "),
    write(N),nl,
    nl.
The Cut (6): Advantages of using the cut

- Program runs faster as alternatives are ignored in the search.
- More space efficient as backtracking points do not need to be recorded.
- The problems with the cut is that the declarative meaning of the program may be changed ...
The Cut (7): Disadvantages of using the cut

- The cut may change the declarative meaning of the program
  
  $p:~ a,~ b.$
  
  $p:~ c.$

  has meaning $p \leftrightarrow (a \land b) \lor c$, and the clause order is irrelevant.

  $p:~ a,!b.$
  
  $p:~ c.$

  has meaning $p \leftrightarrow (a \land b) \lor (\neg a \land c)$, (e.g., $a,~ c: \text{true},~ b: \text{false} \Rightarrow p: \text{false}$), and swapping clause order is relevant.

  $p:~ c.$
  
  $p:~ a!,b.$

  has meaning $p \leftrightarrow c \lor (a \land b)$, thus necessitating dependance on the procedural interpretation of Prolog.

- Cuts that change the declarative meaning (red cuts) of a program should be avoided. Those that do not change the declarative meaning (green cuts) may be used.
The Cut (8): Tutorial

• The following classifies numbers into the classes positive, zero and negative. Rewrite it in a more efficient way using cuts.

    class(Number,positive):-
        Number>0.
    class(0,zero).
    class(Number,negative):-
        Number<0.

• Rewrite the \texttt{gcd} program.
Logical OR (;)

• $a :- b; c$.
  is read procedurally as "to satisfy $a$, satisfy either $b$ or $c$" and declaratively as "$a$ is true if either $b$ or $c$ is true".

• , binds stronger than ;, so
  $a :- b, c; d, e$.
  is equivalent to
  $a :- b, c$.
  $a :- d, e$.
  The latter method of writing is clearer and is definitely preferred.
Database Facts & File Handling (1)

• A file name is a symbol.

• Database facts:
  
  database
  
  person(symbol, char, symbol, symbol)

• asserta(X) adds the clause X to the program before any other clauses with the same head predicate symbol. X must be instantiated so that the head predicate symbol is known. assertz(X) adds X to the program after any clauses with the same head predicate symbol.

  asserta(person(charles, m, xindong, yan)).

• retract(X) removes, from the program, a clause which unifies with X. X must be instantiated so that the head predicate symbol and arity are known. Like assert, retract is not undone on backtracking. retract must be used to remove something from the program. E.g.,

  retract(person(xindong, _, _, _)).
Database Facts & File Handling (2)

- `consult(DosFileName)` reads database facts from the file `DosFileName`.
- `save(DosFileName)` saves all database facts into the file `DosFileName`.
Relational Databases and Prolog

- Prolog is used by relational database developers, using predicates to store and remove data. The relation name is used as the predicate symbol, and each field of a tuple is stored as an argument of a fact. The relational `select` operation is immediately available, with unifiability as the selection criteria, via a query. By allowing rules in the relational database a deductive database is formed.

- These predicates should never be used to store temporary results - use another variable.
Example: Memorizing to Improve Efficiency

fibonacci(1,1) :- !.
fibonacci(2,1) :- !.
fibonacci(Number, Fibonacci):-
  fib(Number, Fibonacci),!.
fibonacci(Number, Fibonacci):-
  Number > 2,!,
  N1 = Number - 1,
  fibonacci(N1, F1),
  N2 = Number - 2,
  fibonacci(N2, F2),
  Fibonacci = F1 + F2,
  asserta(fib(Number, Fibonacci)).
The findall Predicate (1/2)

- It is used for collecting up alternative solutions, as would be provided by asking for more at the prompt.

\[
\text{findall(Variable,Query, List_of_solutions_for_the_variable)}
\]

- For each possible solution of \text{Query}, the values of \text{Variable} are collected together into the \text{List}, in the order in which they are found.
The \texttt{findall} Predicate (2): An Example

lecturer(tim,ai).
lecturer(john,pascal).
lecturer(tim,ai).
lecturer(xindong,ai).
lecturer(john,programming).
lecturer(damien,networks).
lecturer(peter,graphics).
lecturer(xindong,projects).
lecturer(tim,ppl).
lecturer(john,graphics).

\begin{verbatim}
?-findall(Subject,lecturer(Person,Subject),Subject_list).
yes
Subject_list = [ai,pascal,ai,ai,programming,networks,graphics,projects,ppl,graphics]
\end{verbatim}
The bagof Predicate

• bagof(Variable,Query,List_of_solutions_for_the_variable)

• For each combination of solutions for variables other than Variable in the Query, all possible solutions for the Variable are collected together into the List, in the order in which they are found. Thus bagof has itself multiple solutions.

/* The same 10 facts as for the findall example */
--------------------------------------------------------
?-bagof(Subject,lecturer(Person,Subject),Subject_list).
yes
Person = tim
Subject_list = [ai,ai,ppl] ? ;
yes
Person = john
Subject_list = [pascal,programming,graphics] ? ;
yes
Person = xindong
Subject_list = [ai,projects] ? etc, etc
The setof Predicate

- setof(Variable, Query, List_of_solutions_for_the_variable)
- As for bagof, but the list has duplicates removed and is ordered lexicographically.

/* The same 10 facts as for the findall example */
--------------------------------------------------------
?-setof(Subject, lecturer(Person, Subject), Subject_list).
yes
Person = tim
Subject_list = [ai, ppl]  ? ;
yes
Person = john
Subject_list = [graphics, pascal, programming]  ? ;
yes
Person = xindong
Subject_list = [ai, projects]  ? etc, etc
Logic Programming Style (1/4): Layout

• Procedures (clauses for a given predicate symbol and arity) should be grouped.

• Body predicates should be one per line. If there is only one it may be on the same line as the head.

• Procedures should be clearly separated.

• Style must be consistent.

• Distinct modules should be separated into distinct files.
Programming Style (2): Commenting

- The entry level procedures should be documented as such at the top of the program.
- Procedures should be commented as to their overall purpose.
- Clauses should be commented to indicate their specific purpose within the procedure.
Programming Style (3): Clauses

• Should be short.
• Should use meaningful names.
• Recursive procedures should have boundary conditions first and catch all conditions last.
• Cuts and nots should be used with great care, and be well commented.
• Do not use ; for or. Rather write extra clauses.
• There is seldom a need for =. Simply use the same variable.
• Beware of falling foul of the lack of occurs check in most Prologs.
Programming Style (4): Predicates

- Arguments in predicates should be ordered with "input" arguments first, subsidiary data next, and output arguments last.
3. Features of Logic Programming: Review

- Data types and variables
- Expressions and the assignment statement
- Statement-level control structures
- Input and output arguments
- Recursive list processing
3.1. Data Types and Variables

- Terms: All data objects in Prolog are called terms.
  - Atomic terms: 6 types
  - Function terms
  - Composite terms
  - Lists: A head is always an element, and the tail is still a list.
  - No built-in array and pointer types, but you can use lists to implement them.

- Variables
  - Uppercase or _ for start of variables.
  - A variable is either free or bound, and \( X = X + 1 \) is never acceptable.
3.2. Expressions and the assignment statement

- Try to avoid using = (or is) – Simply use the same variable!
- In an evaluation with the = predicate, the RHS must be evaluable, i.e., no unbound variables.
3.3 Statement-Level Control Structures

- **If ... then ... else**: Hidden conditions in different versions of the same clause, especially when `!` is present.
- **Goto** and subprograms: Call a predicate in a rule body, with well-defined arguments.
- **While ... do**: Recursion (with exit points and a recursive body.
  - Empty and single-element lists as input are always good exit points
  - Make sure that the output at each exit point is correct.
3.4 Input and Output Arguments

- Input and output arguments can be interchangable in Prolog, like `sum/3` and `product/3`.
- You should have in mind what are inputs and what is the output before you write your program.
- Always get your “algorithm” or logic designed before writing the code.
FAQ: Why Not $X = X + 1$

• Variables are local to their clause, and each time a clause is used a new set of variables are allocated.

• A variable is either free or bound. Once bound, the value cannot be changed.

• $X$ can be either an input or an output at each run.
  – When as an input, it is bound and cannot be put on the LHS.
  – When as an output, it is unknown and cannot be put on the RHS.
3.5 Recursive List Processing

• Two essential parts for direct recursion:
  – An exit part where the conditions are met. These conditions are normally special cases for input arguments.
  – A recursion body where the conditions/counters are changed.

• Example: The number of generations in `ancestor/3` and `descendant/3`
  – What are the exit points here?
  – How to increase the number of generations?
Example: Sorting Lists

• Suppose \( L \) is a list of integers in arbitrary order, write a set of clauses with head predicate \( \text{sort}(L, \text{Sorted}) \), where \( \text{Sorted} \) is a list of the same items in \( L \) but sorted in the ascending order.

• Exit point(s):

• Recursion body:
Exercise: Binary Search

Write a Prolog program to implement a binary search. Given a list (L) of real numbers that have been organized in ascending order, and a particular number (N), if N exists in L, shows how N is found with the binary search. You can assume that there are no duplicates in the given list L. For example:

?- bsearch([1,2,3,4,5,6,7,8],2).
yes
Try [1,2,3,4] now...
Try [1,2] now...
Try [2] now...
Find 2 in [2]!

Please use the cut statement when it is useful.
Assignment (Option 1): A Family Database System

- Write up a Prolog program to tell 2 persons’ relation.
- Information about an individual person must be stored as a fact containing the person’s name, sex and parents’ names, e.g. `person(charles,m,xindong,yan)`.
- When given 2 names, your program should answer yes if there is a relation between the 2 persons, and the relation (which could be father, mother, sister, brother, son, daughter, aunt, uncle, wife, husband, descendant, or ancestor) should be given as exactly as possible (e.g., sb1 is sb2’s 20th generation ancestor.).
- I will prepare a data file described in the only given predicate and check if your program can give me the expected responses. The file name will be `family.db`.
Assignment (Option 2): A Flight Planning System

• Flight details are stored in the following form:

   flight(Origin, Destination, Dep_time, Arr_time, Week_day, Airline, Flight_num, 
   NFlying_days, Price)

• Write a Prolog program which plans trips with information for the 
  following types of questions:

   – A long trip might require taking several flights in a row. Does a 
     particular airline provide all necessary flights to fly from one city 
     (say Melbourne) to another city (say Tokyo)?
   – To fly from Melbourne to Tokyo, what would be the cheapest trip? 
     And what would be the quickest trip? (Hint: Allow 30 minutes for 
     transitions at each airport).
   – The passenger might like to stop over at certain airports for certain 
     times. This information should be accommodated when planning 
     trips. Also, a good sequence of flights allows for changing planes 
     and does not go to out-of-the-way intermediate destinations. It 
     should stay with the same airline as far as possible.