Logic programming with Prolog

Dr. Constantinos Constantinides, P.Eng.

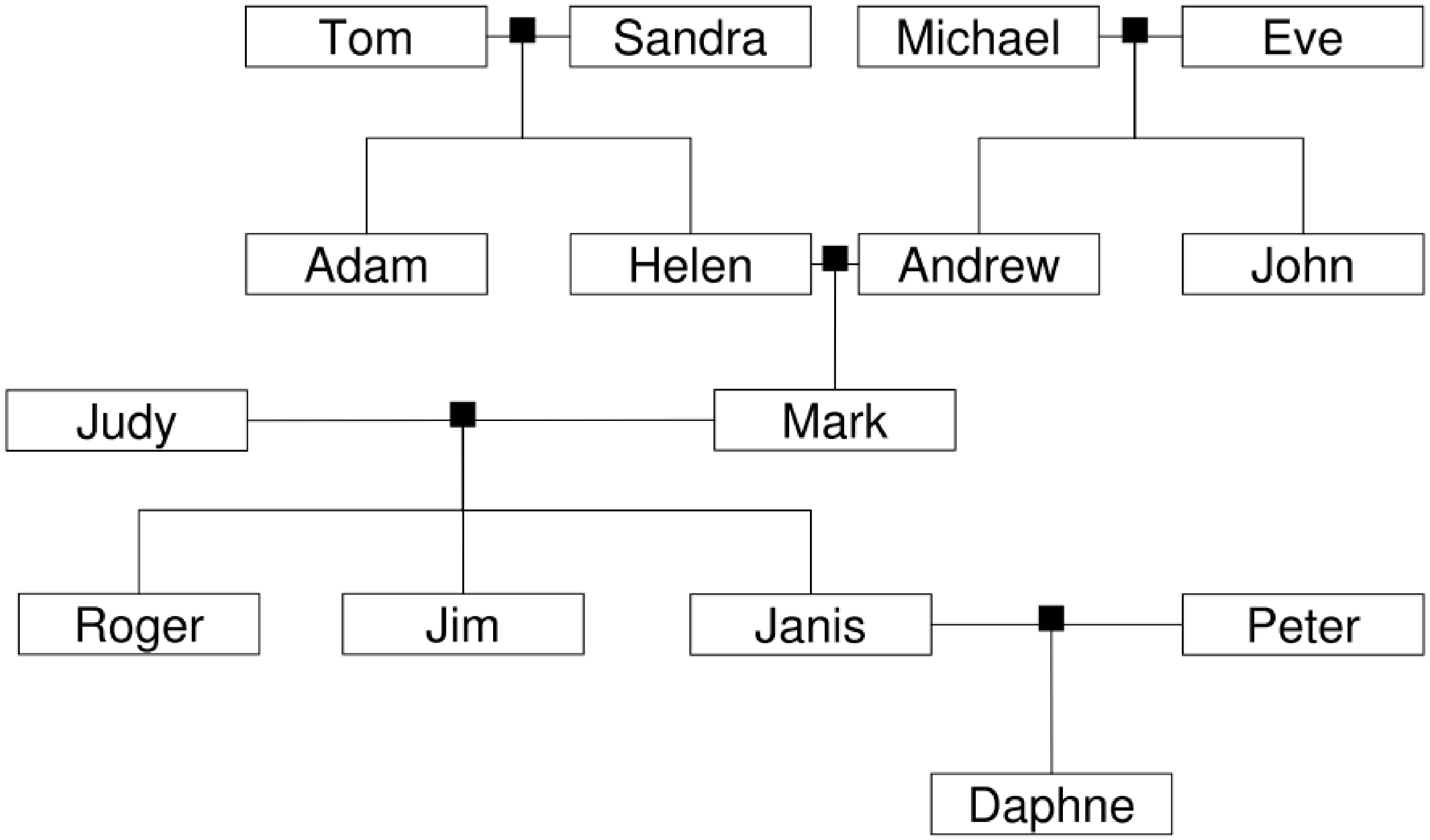
Department of Computer Science and Software Engineering
Concordia University

constantinos.constantinides@concordia.ca

Programming in logic

- Predicate logic can be used to represent and reason about knowledge.
- We will adopt the Prolog programming language to model and process clauses.
- In this discussion we will use a running example to express the meaning and constraints of data as well as to construct queries over their representation in order to obtain information.

A running example: A family genealogy tree



Programs and statements

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

Prolog programs consist of collections of statements (called assertions, or clauses).

Statements and procedures

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

Statements are grouped into procedures.

In the example, we have one procedure named parent, made up of several statements.

Procedures and arguments

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

Each procedure defines a certain relationship between its arguments.

The programmer decides on how to interpret this relationship.

Here, parent(tom, adam) will be interpreted as "*Tom is the parent of Adam*."

Statements revisited: Facts and rules

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

There are two kinds of statements:

1) facts, and

2) rules.

Facts are propositions declared to be true.

In the example, the procedure named parent consists only by facts.

Questions and queries

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

The collection of statements constitutes a (declarative) database.

We can pose queries on this database.

A query is the codification of a question.

There are only two types of queries:

1. *Is it indeed the case that a given statement is true?* (ground query)
2. *Under what conditions, if any, is a given statement true?* (non-ground query)

Ground queries

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

Ground queries result in a Yes/No (or True/False) response:

For example, the question

“*Is it indeed the case that Peter is the parent of Daphne?*”

will be codified into a query and executed as

```
?- parent(peter, daphne).
```

Evaluation of ground queries

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

Prolog will take the query

```
?- parent(peter, daphne).
```

and will start searching the database from top to bottom, one statement at a time trying to match ("unify") it with a statement.

In trying the first statement, a match (unification) is *not* successful.

Evaluation of ground queries /cont.

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

Prolog will then try to match the query

```
?- parent(peter, daphne).
```

against the next statement in the program.

Again, if not successful, it will try the next statement in the program…etc. until either a match is found or until the database is exhausted.

Evaluation of ground queries /cont.

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne). ←
```

In this example, Prolog will eventually succeed, having matched the query

```
?- parent(peter, daphne).
```

with the last statement.

Prolog will respond *Yes* (or *true*) to the query.

We have managed to prove that it is indeed the case that `parent(peter, daphne)` is true.

Non-ground queries

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

A non-ground query involves one or more variables.

The question *"Who is a parent of Daphne"* can be transformed into a non-ground query as

```
?- parent(X, daphne).
```

where X (note the capitalization) is a variable.

We are asking Prolog to seek instantiation(s) for variable X, provided any exist, that could make the query succeed.

Unification revisited

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

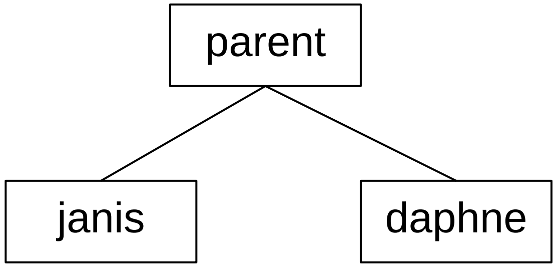
Unification (or matching) is a basic operation on terms.

A ground query can unify with a statement,e.g.

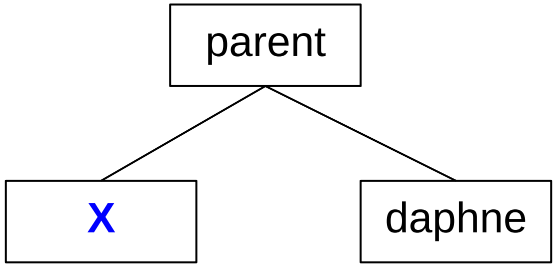
```
?- parent(tom, adam).
```

A non-ground query can unify with a statement only if substitution can be made for any variables so that the two terms can be made equal. In this example, we have

```
?- parent(X, daphne).
```

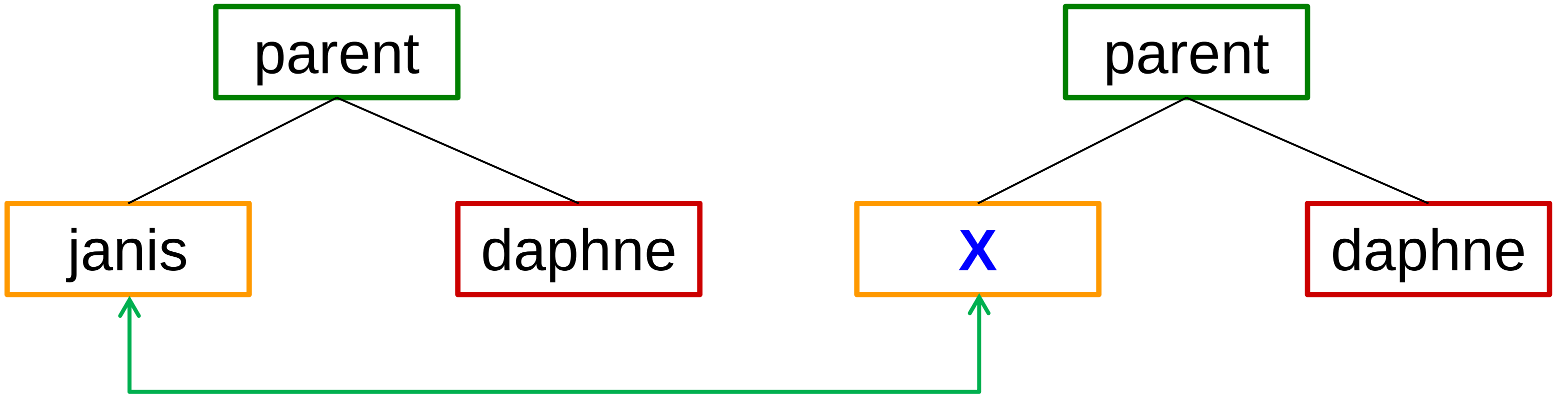


Tree representation of a statement.



Tree representation of a non-ground query.

Unification revisited /cont.



- The terms `parent(janis, daphne)` and `parent(X, daphne)` unify, instantiating `X` to janis.

- There is in fact one more solution, because there are two possible choices to unify with `parent(X, daphne)`.

Rules: Head and body

A rule is an implication between propositions.

The general form is

```
 head :- body.
```

which reads

“The head (of the rule) is true, if the body is true.”,

or, alternatively:

“The head of the rule can succeed if the body of the rule can succeed.”

Formulae and rules

- Let us extend the database with a new procedure grandparent.

- Let p stand for the binary relation isParentOf and let g stand for the binary relation isGrandParentOf.

- We can define g in terms of p by the following formula: For persons x, y, z:

$$G = \forall x, y, z((p(x, z) \wedge p(z, y)) \rightarrow g(x, y))$$

- We can represent the formula with the rule below:

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

Rules and goals

Consider the following rule :

$$\underbrace{\text{grandparent(X, Z)}}_{\text{head}}\ \text{:-}\ \underbrace{\overbrace{\text{parent(X, Y)}}^{\text{goal}},\ \overbrace{\text{parent(Y, Z)}}^{\text{goal}}.}_{\text{body}}$$

The body of rule grandparent/2 contains two goals.

The goals are related by a conjunction
(denoted by the comma symbol).

The rule is now added to the database.

Evaluation of a ground query in the presence of rules

An example

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
```

Consider the query `grandparent(judy, daphne).`

Prolog will search its database from top to bottom, and

a) **unify** the query with the head of the rule,

b) **instantiate** X to judy and Z to daphne,

$$\begin{tikzcd}
\text{grandparent} \arrow[d, leftrightarrow] & \text{(} & X \arrow[d, leftrightarrow] & \text{,} & Z \arrow[d, leftrightarrow] & \text{) :- parent(X, Y), parent(Y, Z).} \\
\text{?- grandparent} & \text{(} & \text{judy} & \text{,} & \text{daphne} & \text{).}
\end{tikzcd}$$

Evaluation of a ground query in the presence of rules
An example /cont.

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
```

c) **resolve** to two new queries (that correspond to the two goals of the rule):

```
parent(judy, Y), parent(Y, daphne).
```

Both queries must now be evaluated (in the order specified) and if both prove true, then <u>the rule succeeds </u>(and the answer to the query is Yes/True).

Evaluation of a ground query in the presence of rules

An example /cont.

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
```

The two goals

```
parent(judy, Y), parent(Y, daphne).
```

will be evaluated as follows:

The first goal parent(judy, Y), will be executed as any other query, unifying with parent(judy ,roger), and instantiating Y to roger.

Once the first goal succeeds, Prolog will try the next one on the right, for the same instantiation:

Can roger make the second goal succeed?

No. The query parent(roger, daphne) is not successful.

Evaluation of a ground query in the presence of rules

An example /cont.

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
```

Prolog will continue searching the database to find matches that can satisfy both goals

```
parent(judy, Y), parent(Y, daphne).
```

The first goal, parent(judy, Y), can unify with parent(judy, jim), instantiating Y to jim.

Can jim make the second goal succeed?

No. The query parent(jim, daphne) is not successful.

Evaluation of a ground query in the presence of rules

An example /cont.

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
```

Prolog will continue searching the database to find matches that can satisfy both goals

```
parent(judy, Y), parent(Y, daphne).
```

The first goal, parent(judy, Y), can unify with parent(judy,janis), instantiating Y to janis.

Can janis make the second goal succeed?

Yes. The query parent(janis, daphne) is successful.

Evaluation of a non-ground query in the presence of rules: An example

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
```

The question

“Who are the grandparents of Daphne?”

is codified into the query

```
?- grandparent(G, daphne).
```

This will unify with the head of rule grandparent, instantiating variable Z to daphne and resolving into two goals:

```
parent (G, Y), parent(Y, daphne).
```

Evaluation of a non-ground query in the presence of rules: An example

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(eve, andrew).
parent(eve, john).
parent(helen, mark).
parent(andrew, mark).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
grandparent(X, Z) :- parent(X, Y),
                     parent(Y, Z).
```

```
?- grandparent(X, daphne).
X = judy ;
X = mark ;
No
```

Upon finding a match, Prolog will stop here and wait for instructions. The semicolon symbol (;) inquires whether more matches can be found.

A period symbol (.) would indicate our intention to stop the search.

Multi-line rules: Disjunction

- We can now further extend the database with a rule to define the binary relation “is ancestor of.”
- Suppose we let p stand for the binary relation isParentOf and let a stand for the binary relation isAncestorOf relation.
- We can now define a in terms of p by the following formula we will call A:

$$A = \forall x, y(p(x, y) \ \rightarrow \ a(x, y))$$

$$A = \forall x, y, z((p(x, z) \ and \ a(z, y)) \rightarrow a(x, y))$$

Multi-line rules: Disjunction /cont.

- In other words, x is an ancestor of y if either x is a parent of y, or x is a parent of an ancestor of y. We can represent this in Prolog with the following multi-line rule:

$$\text{disjunction}\begin{cases}\text{ancestor(X, Y) :- parent(X, Y).}\\ \text{ancestor(X, Y) :- }\underbrace{\text{parent(X, Z), ancestor(Z, Y)}}_{\text{conjunction}}\text{.}\end{cases}$$

- A rule can be placed in more than one lines.

- In this case, there is a <u>disjunction</u> between the two lines, and there is a <u>conjunction</u> between the two goals of the body of the second rule.

Further extending the database

```
man(tom).
man(michael).
man(adam).
man(andrew).
man(john).
man(mark).
man(roger).
man(jim).
man(peter).
woman(sandra).
woman(eve).
woman(helen).
woman(judy).
woman(janis).
woman(daphne).
father(X, Y) :- man(X),
                parent(X, Y).
mother(X, Y) :- woman(X),
                 parent(X, Y).
```

We extend the database by introducing four procedures:

man, woman: made up of facts,

and

father, mother: made up of rules.

```
father(X, Y)   :- man(X),
                  parent(X, Y).

mother(X, Y) :- woman(X),
                parent(X, Y).
```

Anonymous variables in rules

- If any parameter of a relation is not important, we can replace it with an anonymous variable denoted by the underscore character (_) as follows:

```
is_father(X) :- father(X, _).
is_mother(X) :- mother(X, _).
```

- We can now pose more queries such as *“Is Tom a father?”*

- To answer this type of question, it does not matter who Tom is the father of, as long as Tom is found as the first term in a father statement. The query is as follows:

```
?- is_father(tom).
true
```

Anonymous variables in queries

- Alternatively we can use anonymous variables in queries, such as

```
?- father(tom, _).
true
```

Lists

- Lists are represented in square brackets […].
- The empty list is represented by [].
- Every non-empty list can be represented in two parts:
  - The head, which is the first element.
  - The tail, which is the list containing the remaining elements.

  - The head of [john, eve, paul] is john.
  - The tail of [john, eve, paul] is [eve, paul].

List representations

- The symbol | in [H|T] represents a list whose head is H and whose tail is T.
- We can represent the above example as [john | [eve, paul]]
- Since [eve, paul] is also a list with head eve and tail [paul], we can write the above list as [john | [eve | [paul]]]
- Any one-element list can be written as that element joined to the empty list. Thus, [paul] is the same as [paul | []]
- We can now write the full list as [john | [eve | [paul | [ ]]]]

Example: Checking for list membership

- We want to define a procedure member/2 which succeeds if an element is found inside a list.

- We can define list membership recursively as follows:

- Element X is a member of a list L if

  X is the head of L (regardless of what the tail is), or

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

  or

  X is a member of the tail of L (regardless of what the head is).

```
member(X, [_|T]) :- member(X, T).
```

Example: Checking for list membership /cont.

- The rule is now

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

member(X, [_|T]) :- member(X, T).
```

Example: Checking for list membership /cont. Alternative implementation using the ‘cut’ operator

- An alternative implementation of member/2 is shown:

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

member(X, [_|T]) :- member(X, T).
```

The ! tells the interpreter to stop looking for alternatives once the LHS succeeds.

However, if `member(X, [X|_])` fails it will check `member(X, [_|T])`.

Example: Checking for list membership /cont.

Initial implementation

```
?- member(a, [a, b, c]).
true

?- member(a, []).
false.

?- member([b, c], [[a], [b, c]]).
true

?- member(X, [a, b]).
X = a ;
X = b ;
false.
```

Implementation using ‘cut’

```
?- member(a, [a, b, c]).

?- member(a, []).
false.

?- member([b, c], [[a], [b, c]]).

?- member(X, [a, b]).
X = a ;
false.
```

Example: The last element in a list

- In this example, we want to define a rule last/2 which succeeds if an element is found to be in the last position of a non-empty list.

- We can identify two cases for this:

1. The list has one element.

2. The list has more than one element.

Example: The last element in a list /cont.

- Case 1: The list has only one element.
- In this case, the last element is the only existing element of the list.
- The following rule,

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

reads *“The rule succeeds if element L is found to be the only element of a given list.”*

Example: The last element in a list /cont.

- Case 2: The list has more than one element.
- In this case, we need to reduce the problem to the one that can be handled by case 1.
- In other words, the clause will succeed once it chops off all elements, one by one, until it ends up with one element.

Example: The last element in a list /cont.

- The following rule,

```
last(L, [H|T]) :- last(L, T).
```

  reads *“The rule can succeed for a list whose head is H and whose tail is T, if it can succeed for a new list which is the tail T of the original list.”*

Example: The last element in a list /cont.

- In other words, let us get rid of the first element and see if we end up with only one element in which case the rule of case 1 will determine that this remaining element is indeed the last element.

- However, if after getting rid of the first element we end up with a list with more than one elements, we must repeat this chopping off the head of the list, until we end up with a list which has only one element and subsequently handled by the first rule (of case 1).

Example: The last element in a list /cont. Evaluation of a ground query [1 of 3].

Given the rule,

```
last(L, [L]).
last(L, [H|T]) :- last(L, T).
```

Consider the query

```
?-  last(c, [a, b, c]).
```

Prolog will search its database (from top to bottom) and

unify the query with the head of the second statement of the rule,

instantiate variable L to c and variable [H|T] to [a | b, c],

resolve to a new query (that corresponds to the body of the rule):

last(c, [b, c]).

This new query must now be evaluated.

Example: The last element in a list /cont. Evaluation of a ground query [2 of 3].

Given the rule,

```
last(L, [L]).
last(L, [H|T]) :- last(L, T).
```

The query

```
?- last(c, [b, c]).
```

Will cause Prolog to (perform a new search and)

unify the query with the head of the second statement of the rule,

instantiate variable L to c and variable [H|T] to [b | c],

resolve to a new query (that corresponds to the body of the rule):

```
last(c, [c]).
```

This new query must now be evaluated.

Example: The last element in a list /cont. Evaluation of a ground query [3 of 3].

Given the rule

```
last(L, [L]).
last(L, [H|T]) :- last(L, T).
```

The query

```
?-  last(c, [c]).
```

Will cause Prolog to (perform a new search and)

unify the query with the head of the first statement of the rule, and yield a success, indicating that it is indeed the case that `c` is found in the last position of the list.

Arithmetic, relational and logical operators

- Available arithmetic operators: +, -, *, /, **, mod.
- Keyword is denotes an assignment.
- Relational operators: <, >, =<, >=, =:=, =\=

  = is used for unification, e.g. X = 7

  == is used to denote identical terms, e.g. 5 == 5

  is denotes an assignment and evaluates the RHS, e.g. X is (1 + 2)

  =:= evaluates both sides, e.g. (1 + 3) =:= (2 + 2)

- Logical operators: \+

Example: Find the largest between two numbers

<u>Correct (using ‘cut’)</u>

```
larger(X, Y, X) :- X > Y, !.
larger(X, Y, Y).

?- larger(5, 7, X).
X = 7 .

?- larger(11, 7, X).
X = 11 ;
false.
```

<u>Incorrect</u>

```
larger(X, Y, X) :- X > Y.
larger(X, Y, Y).

?- larger(11, 7, X).
X = 11 ;
X = 7 ;
false.
```

Example: Calculating the size of a list

- We can recursively obtain the length of a list as follows:

- If the list is empty, then its length is 0.
- If the list is not empty, then its length is 1 + length of its tail.

- Consider rule size/2 to read in a list and calculate its length.

```
size([],0).
size([H|T],N) :- size(T,N1), N is N1+1.
```

Example: Calculating the size of a list

- We can execute queries as follows:

```
?- size([],N).
N = 0.

?- size([a,b,c],N).
N = 3.

?- size([[a,b],c],N).
N = 2.

?- size([[a,b,c]],N).
N = 1.
```

Example: Calculating the size of a list

- We can execute queries as follows:

```
?- delete(a, [a], NewList).
NewList = [] .

?- delete(a, [a, e, o], NewList).
NewList = [e, o] .

?- delete(a, [e, o, a], NewList).
NewList = [e, o] .
```

Example: Deleting an element from a list

- Consider a list L and an element X that we need to delete from L.

- We can identify three cases:

    1. If X is the only element, then the result is the empty list.

    2. If X is head of L, then the result is the tail of the initial list.

    3. If X is present in the tail part, then keep the head and recur on the tail.

- Consider rule `delete/3`:

```
delete(X, [X], []).
delete(X,[X|L1], L1).
delete(X, [Y|L2], [Y|L1]) :- delete(X,L2,L1).
```

Built-in utility functions

- The built-in function `findall(X, P, L)` returns a list `L` with all values for `X` that satisfy predicate `P`.
- To eliminate redundancies in a list, we can use the built-in function `list_to_set(List, Set)` that converts the list (with possibly repeated elements) into a set.
- The built-in function `length(List, L)` returns the length `L` of a given list.

Example with **findall** and **list_to_set** in a query

- Let us obtain a set of all fathers:

```
?- findall(F, father(F, _), Lst).
Lst = [tom, tom, michael, michael, andrew, mark,
       mark, mark, peter].

?- findall(F, father(F, _), Lst), list_to_set(Lst, Set).
Lst = [tom, tom, michael, michael, andrew, mark,
       mark, mark, peter],
Set = [tom, michael, andrew, mark, peter].
```

Example with **findall** and **list_to_set** in a rule

- The query

  `findall(F, father(F, _), Lst), list_to_set(Lst, Set)`

  is rather long and complex.

- We can encapsulate its size and complexity in a rule `get_all_fathers/1`:

```
get_all_fathers(Set) :-
    findall(F, father(F, _), Lst),
    list_to_set(Lst, Set).
```

```
?- get_all_fathers(Set).
Set = [tom, michael, andrew, mark, peter].
```

Example with **findall** and **length** in a rule

- Let us construct a rule `qualifies_for_benefits/1` that succeeds if the argument is a mother of at least three children.

```
qualifies_for_benefits(P) :-
    woman(P),
    findall(P, parent(P, _), L),
    length(L, N),
    N >= 3.
```

```
?- qualifies_for_benefits(Name).
Name = judy ;
false.
```