**CSCI: 4500/6500 Programming Languages**

**Prolog & Logic Programming**

Maria Hybinette, UGA

---

**Prolog Download**

**Binaries and Source**

- SWI-prolog (swipl 5.10.4-6.0.2 depending on platform) website:
  - Mac OS X on Intel & PPC (Tiger, Leopard (46.3 MB), Snow Leopard and Lion binaries available)
  - Linux RPMs.
  - Windows NT, XP, Vista7, 2000, 64 Bit,
  - Source Install
- XQuartz (X11) 2.5.0 for help & development tools.

---

**Great Prolog Tutorials**

- JR Fisher’s original tutorial:
- Roman Barták’s interactive tutorial:
- Mike Rosner’s crash course:
- James Lu and Jerud Mead’s tutorial:
  - [http://www.csc.ucsc.edu/classes/cmps112/spring03/languages/prolog/PrologIntro.pdf](http://www.csc.ucsc.edu/classes/cmps112/spring03/languages/prolog/PrologIntro.pdf)
- James Power’s tutorial:
  - (2012 not available – BUT let me know if you find it – it is a good one)

---

**What is Prolog?**

- A declarative or logic programming language
  - Specifies the results (describes what the results look like)
  - In contrast to a "procedure" on how to produce the results.
- Based on first order predicate calculus
  - Consists of propositions that may or may not be true
- Prolog uses logical variables
  - Not the same as variables in other languages
  - Used as ‘holes’ in data structures that are gradually filled in as the computation processes (will see examples)

---

**What is Prolog?**

- Alain Colmeraure & Philippe Roussel, 1971-1973
  - With help from theorem proving folks such as Robert Kowalski
  - Colmeraure & Roussel wrote 20 years later:
    *Prolog is so simple that one has the sense that sooner or later someone had to discover it... that period of our lives remains one of the happiest in our memories.*

---

**Let’s look at a sample session...**

```
(saffron:ingrid:815) swipl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.9)
Copyright (c) 1990-2006 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software, and you are welcome to redistribute it under certain conditions. Please visit http://www.swi-prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).
?- ['second'].
% first compiled 0.00 sec, 594 bytes
```
Look at a sample of code...

```prolog
second.pl

elephant(kyle). % this is a comment
elephant(kate).
panda(chi_chi).
panda(ming_ming).
dangerous(X) :- big_teeth(X).
dangerous(X) :- venomous(X).
guess(X,tiger) :- striped(X), big_teeth(X), isaCat(X).
guess(X,koala) :- arboreal(X), sleepy(X).
guess(X,zebra) :- striped(X), isaHorse(X).
```

Prolog Programs are “Declarative”

I declare that the leaves are green and elephants are mammals.

- Clauses are statements about what is **true** about the problem (as statements and questions).
  - instead of instructions on how to accomplish the solution.
- Prolog finds answers to **queries** by parsing through “the database” of possible solutions.

Anatomy of Prolog

**Declarative Component:** “the program” (“the Database”):

- Consists of **facts** and **rules**
- Defines the relations on sets of values

**Imperative Component**: “the execution engine”, the “Prolog Solver”:

- extracts the sets of data values **implicit** in the facts and rules of the program
- **Unification** - matching query and “head” of rules (later)
- **Resolution** - replaces the head with the body of the rule and then applies substitution to form a new query(ies).

Syntax of Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Upper Case _ Denotes Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constant</strong></td>
<td>Names an individual</td>
</tr>
<tr>
<td>Atom</td>
<td>Names an individual that has parts</td>
</tr>
<tr>
<td>Number</td>
<td>Stands for an individual unable to be named when program is written</td>
</tr>
<tr>
<td>flexible</td>
<td>-7</td>
</tr>
<tr>
<td>spaghetti</td>
<td>3</td>
</tr>
<tr>
<td>super</td>
<td>1.658</td>
</tr>
<tr>
<td>califragilistic</td>
<td>2.45e-27</td>
</tr>
<tr>
<td>gakemon</td>
<td>-13.6</td>
</tr>
</tbody>
</table>

**Compound Term**

- **Atom**
- **Variable**

constant versus Variables

- **Variables** start with a capital letter, A, B, ... or underscore _:
  - Food, Person, Person2, _A123
- **Constant** “atoms” start with a, b, ...z or appear in single quotes:
  - maria, olives, isaac, ’CSCI4500’
- Other kinds of constants besides atoms:
  - Integers -7, real numbers 3.14159, the empty list []
- **Note**: Atom is not a variable; it is not bound to anything, never equal to anything else
**Constant versus Variables**

- Nothing stops you from putting constants into constraints:

  % what Food does eric eat?
  eats(eric, Food).
  % 2 answers: chips & pear
  % use ';' for next answer.

  % what Person eats fish?
  eats(Person, fish).
  % 2 answers: ? & …? ...

  % who will share what with robert? ** more later
  eats(robert, Food), eats(Person, Food).

  Try it!

  eats(adam, sushi).
eats(eric, chips).
eats(eric, pears).
eats(isaac, fish).
eats(isaac, fish).
eats(ibti, chips).
eats(ibti, sushi).
eats(jordan, fish).
eats(jordan, olives).
eats(jonathan, olives).
eats(jonathan, chips).
eats(maria, sushi).
eats(robert, olives).
eats(young, olives).
eats(young, pears).

**Familiar Compound Terms**

- The parents of Spot and Fido and Rover

  parents(spot, fido, rover)

  `parents` (atom) of arity 3.

  Components (any terms)

- Can depict the term as a tree

  `parents` `spot` `fido` `rover`

**Summary Terms**

- All Prolog programs and data are built from such terms

- Later, we will see that, for instance, +(1,2) is usually written as 1+2

- But these are not new kinds of terms, just abbreviations

```prolog
<xterm> ::= <constant> | <variable> | <compound-term>
<constant> ::= <integer> | <real number> | <atom>
<compound-term> ::= <atom> ( <termlist> )
<termlist> ::= <term> | <term> , <termlist>
```

**The Prolog Program (Database)**

- A Prolog language system maintains a collection of facts and rules of inference

- It is like an internal database

- A Prolog program is just a set of data for this database

- The simplest kind of thing in the database is a fact: a term followed by a period

**SWI-Prolog**

- Prompting for a query with ?-

- Normally interactive: get query, print result, repeat
The consult Predicate

- Predefined predicate to read a program from a file into the database
  » Example: File eats.pl defines the "eats" constraints, or lists of facts.

Simple Queries

- A query asks the language to prove something
- The answer will be True or False
- Some queries, like consult are executed only for their side effects.
- Example Query program:
  » Does kyle eat fish (type query)?

Simple Queries: the Period '.'

- Queries can take multiple lines
- If you forget the final period, Prolog prompts for more inputs with |.

Simple Queries: the Period ' . '

- Queries can take multiple lines
- If you forget the final period, Prolog prompts for more inputs with |.

Queries With Variables

- Any term can appear as a query, including a term with variables
- The Prolog system shows the bindings necessary to prove the query

Here, it waits for input. We hit Enter (or ;) to make it proceed.
Multiple Solutions

- There might be more than one way to prove the query
- By typing ; rather than Enter, you ask the Prolog system to find more solutions
  - Example: What does kyle eat?

```
?- eats(issac,X).
X = fish ;
X = chips .
```

```
No
```

Conjunctions

- A conjunctive query has a list of query terms separated by commas
  - think of commas as "AND"s
- The Prolog system tries prove them all (using a single set of bindings)
- Example: Query folks that eat common foods with eric

```
?- eats(eric, Food), eats(Person, Food).
Food = chips
Person = eric;
Food = chips
Person = issac;
```

```
```

Flexibility

- Normally, variables can appear in any or all positions in a query:
  - `eats(X,olives)`
  - `eats(corey,X)`
  - `eats(X,Y)`
  - `eats(X,X)`

```
?- eats(adam, sushi).
eats(eric,chips).
eats(eric,pears).
eats(isaac,fish).
eats(isaac,fish).
eats(ibti,chips).
eats(ibti, sushi).
eats(jordan,fish).
eats(jordan,olives).
eats(jonathan,olives).
eats(jonathan,chips).
eats(maria, sushi).
eats(robart, olives).
eats(esse, sushi).
eats(sean,chips).
eats(young,olives).
eats(young,pears).
```

More General Queries

- Query folks that eat common foods:
  - conjoin two constraints with a common food.
  - conjoined with a comma (read as "and").

```
?- eats(Person1,Food),eats(Person2,Food).
Person1 = adam
Food = sushi
Person2 = maria;
```

```
```

More Examples: Conjunctions

```
Sven, Emy
Emil
Ingrid
Knut
Maria
Gunnar
Tucker
Emmy

mariafamily.pl

parent[maria,gunnar].
parent[maria,emmy].
parent[emmy,ingrid].
parent[even,ingrid].
parent[even,emil].
```

```
Great grandchildren of Emy?

1) Who is a child of Emy?
2) Who is a child of ?
3) Who is a child of ?

```

```
```

More Examples: Conjunctions

```
Sven, Emy
Emil
Ingrid
Knut
Maria
Gunnar
Tucker
Emmy

mariafamily.pl

parent[maria,gunnar].
parent[maria,emmy].
parent[emmy,ingrid].
parent[even,ingrid].
parent[even,emil].
```
More Examples: Conjunctions

Great grandchildren of Emy?

?- parent(emy,Child),
  | parent(Child,Grandchild),
  | parent(Grandchild,GreatGrandchild).

Child = ingrid
Grandchild = maria
GreatGrandchild = gunnar ;

Child = ingrid
Grandchild = maria
GreatGrandchild = tucker ;

Child = ingrid
Grandchild = maria
GreatGrandchild = emmy ;

No ?–

Motivation: Need Rules

% Great grandchildren of Emy?
?- parent(emy,Child),
  | parent(Child,Grandchild),
  | parent(Grandchild,GreatGrandchild).

Sven, Emy
Emil
Ingrid
Knut
Maria
Gunnar
Tucker
Emmy

Great grandchildren of Emy?

?- parent(emy,Child),
  | parent(Child,Grandchild),
  | parent(Grandchild,GreatGrandchild).

A Rule

head

\[ \text{greatgrandparent}(GGP, GGC) \] :-
parent(GGP, GP),
parent(GP, P),
parent(P, GGC).

A rule says how to prove something: to prove the head, prove its conditions
To prove \( \text{greatgrandparent}(GGP, GGC) \), find some \( GP \) and \( P \) for which you can prove parent(GGP, GP), then parent(GP, P) and then finally parent(P, GGC)

A Rule

head

\[ \text{greatgrandparent}(GGP, GGC) \] :-
parent(GGP, GP),
parent(GP, P),
parent(P, GGC).

A rule says how to prove something: to prove the head, prove the conditions
To prove \( \text{greatgrandparent}(GGP, GGC) \), find some \( GP \) and \( P \) for which you can prove parent(GGP, GP), then parent(GP, P) and then finally parent(P, GGC)

A Rule

head

\[ \text{greatgrandparent}(GGP, GGC) \] :-
parent(GGP, GP),
parent(GP, P),
parent(P, GGC).

A rule says how to prove something: to prove the head, prove the conditions
To prove \( \text{greatgrandparent}(GGP, GGC) \), find some \( GP \) and \( P \) for which you can prove parent(GGP, GP), then parent(GP, P) and then finally parent(P, GGC)

Facts and Rules

\[ \text{Head} \] :- \text{Body}.
% This is a rule.
\[ \text{Head} \]
% This is a fact.

Head is the consequence.
Head can be concluded if the body is true
Facts and Rules

Head

bioparents(X, Y) :- male(X), female(Y).

Body (pre-conditions)

Note that left side of the rule looks just like a fact, except that the parameters are variables.

Read:
- The pair "parents(X,Y)" satisfies the predicate "parents" if there is a node X and Y such that X satisfies the predicate "X" and "Y" satisfies the predicate Y.

Goals

Clauses

A program consists of a list of clauses.

A clause is either a fact or a rule, and ends with a period.

parent(maria, gunnar).
parent(maria, tucker).
parent(maria, emmy).
parent(ingrid, maria).
parent(ingrid, knut).
parent(maria, ingrid).
parent(anny, ingrid).
greatgrandparent(GGP,, GGC) :-
parent(GGP, GP),
parent(GP, P),
parent(P, GGC).

Interpretation of Clauses

Form of Clause:
H :- G₁, G₂, …, Gₙ.

Declarative Reading:
- "That H is provable follows from goals G₁, G₂, …, Gₙ being provable".

Procedural Reading:
- "To execute procedure H, the procedures called by the goals G₁, G₂, …, Gₙ are executed first".

Example: Clauses: Facts and Rules

Example:
A directed graph of five nodes:

Define the edges of the graph, as facts?

Define a rule called "tedge" which defines the property of a "path of length two" between two edges?

tedge(Node1, Node2) :- edge(Node1, SomeNode),
edge(SomeNode, Node2).

Example 3: Another Rule

Compatible(Person1, Person2) :- eats(Person1, Food),
eats(Person2, Food).

"Person1 and Person2 are compatible if there exists some Food that they both eat."

"One way to satisfy the head of this rule is to satisfy the body"

eats(george, chips). eats(maria, chips). eats(maria, fish).
eats(maria, olives). eats(anny, olives). eats(anny, olives).
Recursive Rules

- **Base case:** \( X \) is a parent of \( Y \).
- **Recursive case:** there is some \( Z \) such that \( Z \) is a parent of \( Y \), and \( X \) is an ancestor of \( Z \).

Prolog tries rules in the order given, so put base-case rules and facts first.

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

Recursion Example 2

- Who’s married to their boss?
  - \( \text{boss}(X,Y), \text{married}(X,Y) \).
- Who’s married to their boss’s boss?
  - \( \text{boss}(X,Y), \text{boss}(Y,Z), \text{married}(X,Z) \).
- Who’s married to their boss’s boss’s boss?
  - Okay, this is getting silly. Let’s do the general case.
  - Who’s married to someone above them?
    - \( \text{above}(X,Y) \).
    - \( \text{above}(X,Y), \text{boss}(X,\text{Underling}), \text{above}(\text{Underling},Y) \).
    - \( \text{above}(X,Y), \text{married}(X,Y) \).

Base case: For simplicity, it says that \( X \) is “above” herself. If you don’t like that, replace base case with \( \text{above}(X,Y) :- \text{boss}(X,Y) \)."

Example: Graph Example

- Embellish graph program to include "paths" of any positive length.
- Thinking Recursively:
  - If there is an edge then there is a path (base)
  - If there is an edge to an intermediate node from which there is a path to the final node.
- Two rules with the same head, reflects logical "or"
- Predicate of head of second rule, is also in the body of that rule.
- These rules together illustrate recursion in Prolog!

```
path(N1,N2) :- edge(N1,N2).
p path(N1,N2) :- edge(N1,N2), path(SomeN,N2).
```

Core Syntax of Prolog

- You have seen the complete core syntax
- There is not much more syntax for Prolog than this: it is a very simple language
- Syntactically, that is!

How does Prolog Compute?

- Deduce useful implicit knowledge from the "program" or data base.
- Computations in Prolog is facilitated by the query, a conjunction of atoms.
- New example (more complicated) program:

```
edge(a,b).
edge(a,e).
edge(b,c).
edge(c,a).
edge(b,d).
edge(e,b).
tedge(N1,N2) :- edge(N1,SomeN),edge(SomeN,N2).
path(N1,N2) :- edge(N1,N2).
path(N1,N2) :- edge(N1,SomeN),path(SomeN,N2).
```
edge(a,b).

- Iterates *in order* through the program’s “edge” clauses.
- *Ground Query*: only value identifiers as parameters to the predicate.
- First one to match is `edge(a,b),` so Prolog returns with `true` (so yes).

edge(b,c).
edge(a,e).
edge(c,a).
edge(b,d).
edge(e,b).

```prolog
tedge(N1,N2) :- edge(N1,SomeN), edge(SomeN,N2).
path(N1,N2) :- edge(N1,N2).
path(N1,N2) :- edge(N1,SomeN), path(SomeN,N2).
```

```prolog
edge(a,b).
path(a,b).
```

- *another ground query*
- No rule that exactly match the query.
- Know, the head is true if the body is true
  - If variable’s N1 and N2 are replaced by a and b, then body of 8 is true
    - `edge(a, b)` is a fact!
    - and the head with the same substitution must be true
  - Prolog conclude that the query is `true`

edge(a,b).
path(a,b).
Unification

- Pattern-matching using Prolog terms
- Two terms unify if there is some way of binding their variables that make them identical.
  - Usually the two terms
    - one from the query (or another goal) and
    - the other being a fact or a head of a rule
- Example:
  - parent(adam,Child) and parent(adam,seth)
  - Do these unify?
  - Yes! they unify by binding the variable Child to the atom seth.

Resolution

- The hardwired inference step
- A clause is represented as a list of terms (a list of one term, if it is a fact)
- Resolution step applies one clause, once, to make progress on a list of goal terms

Resolution

- When an atom from the query has unified with the head of a rule (or a fact),
- Resolution replaces the atom with the body of the rule (or nothing, if a fact) and
- then applies the substitution to the new query.

Resolution

- Unify:
  - tedge(a,X) and tedge(N1,N2).
  - giving the substitution
    - N1 = a, X = N2
- Resolution:
  - replaces tedge(a,X) with body edge(N1,SomeN), edge(SomeN,N2) and apply the substitution above to get the new query.
  - edge(a,SomeN),edge(SomeN,N2)
- Select first atom, edge(a,SomeN)
- Unify:
  - edge(a,SomeN) with edge(a,b),
  - giving the substitution
    - SomeN = b
- Resolution: replace edge(a,SomeN) ...

Backtracking

- There are other solutions, we could redo the computation above and get substitution
  - $X=b$ or $X=c$ or $X=d$
- When Prolog reduces a query to the empty query,
  - it backtracks to the most recent unification to determine whether there is another fact or rule with which the unification can succeed.
  - Backtracking continues until all possible answers are determined.
Recursive Queries

\[
\text{above}(X, X) .
\]

\[
\text{above}(X, Y) :- \text{boss}(X, \text{Underling}), \text{above}(\text{Underling}, Y).
\]

- above(c,h).  % should return True
  » matches above(X, X)? no
  » matches above(X, Y) with X=c and Y=h
    » \text{boss}(c,Underling),
    » matches boss(c,f) with Underling=g
    » above(g,h).
    » ... ultimately fails because g has no underlings...

Review: Basic Elements of Prolog

- **Variable**: any string of letters, digits, and underscores beginning with an **Upper-case** letter
- **Instantiation**: binding of a variable to a value
  » Lasts only as long as it takes to satisfy one complete goal
  » allows unification to succeed
- **Predicates**: represents atomic proposition
  \[ \text{functor}(\text{parameter list}) \]

Review Prolog

- **Prolog program**: Set of propositions
  » **Facts**
  » **Rules**: consequence \( \text{if antecedent is true then the consequence is true} \)
  » \text{edge}(A, B) :- \text{edge}(A, X), \text{edge}(X, B) .
- **Running a program**: A Prolog query (sometimes called goals): A proposition of which truth is to be determined.
  » **Idea**: Prove truthfulness (or "cannot determine" (not falsehood)) by trying to find a chain of inference rules and facts (inference process)
  » **Resolution**: Process that allows inferred propositions to be computed from given propositions
  » Unification merges compatible statements. Binding process.

Inference Process

- **Backward Chaining**, Top-down resolution:
  » Start with goal (query), see if a sequence of propositions leads to set of facts in the database (Prolog)
  » Looks for something in the database that unify the current goal,
  » finds a fact, great it succeeds!
  » If it finds a rule, it attempts to satisfy the terms in the body of the rule (these are now subgoals).
- **Forward Chaining**, Bottom-up resolution:
  » Begin with program of facts and rules in the database and attempt to find a sequence that leads to goal (query).
Backward Chaining

- When goal has more than one sub-goal, can use either
  - Depth-first search: find a complete proof for the first sub-goal before working on others (Prolog)
    - Push the current goal onto a stack,
    - make the first term in the body the current goal, and
    - prove this new goal by looking at beginning of database again.
    - If it proves this new goal of a body successfully, go to the next goal in the body. If it gets all the way through the body, the goal is satisfied and it backs up a level and proceeds.
  - Breadth-first search: work on all sub-goals in parallel

Backtracking

- If a sub-goal fails:
  - reconsider previous subgoal to find an alternative solution
- Begin search where previous search left off
- Can take lots of time and space because may find all possible proofs to every sub-goal

Compound Terms

- Basic blocks: variables, constants and variables
- Compound terms: Seen it already -- it is the functor( parameter list ) structure (e.g., eats( cole, fish ) )
  - Variables cannot be used for the functor
  - However the "parameter list" can be any kind of term (it can be another functor).
  - book( title(lord_of_the_rings), author(tolkien) )
  - Uh uh what about unification now! (matching of goals and heads).

Unification Rules

- Two terms unify:
  - if substitution can be made for any variables in the terms so that terms are made identical.
  - if no such substitution exists, the terms do not unify.
- The unification algorithm proceeds by recursively descent of the two terms.
  - Constants unify if they are identical
  - Variables unify with any term, including other variables
  - Compound terms unify if their functors and components unify

Unification Compound Terms

- Compound terms unify if their functors and components unify (how do terms become equal?)
  - $f(X, a(b, c))$ and $f(d, a(Z, c))$ do unify.

Example 2

- The terms $f(X, a(b, c))$ and $f(Z, a(Z, c))$ unify

- $Z$ co-refers within the term. Here, $X/b, Z/b$.
  - Earlier : $f(X, a(b, c))$ and $f(d, a(Z, c))$ did unify...
What about?

- \( f(c, a(b, c)) \) and \( f(z, a(z, c)) \)?

- No matter how hard you try, these terms cannot be made identical by substituting terms for variables.

Unify?

- \( g(z, f(A, 17, B), A+B, 17) \) and \( g(C, f(D, D, E), C, E) \)?

- First write in the co-referring variables.

- Recursive descent: We go top-down, left-to-right, but the order does not matter as long as it is systematic and complete.

- recursive descent We go top-down, left-to-right, but the order does not matter as long as it is systematic and complete.
• recursive descent We go top-down, left-to-right, but the order does not matter as long as it is systematic and complete.

\[ Z/C, \ C/Z, \ A/17, \ D/17, \ B/E, \ E/B \]
Can also use “substitution method”

Exercise – Alternative Method

Z/C

Exercise – Alternative Method

A/D, Z/C

Exercise – Alternative Method

D/17, A/D, Z/C

Exercise – Alternative Method

D/17, A/17, Z/C
Exercise – Alternative Method

B/E, D/17, A/17, Z/C

Exercise – Alternative Method

B/E, D/17, A/17, Z/C

Exercise – Alternative Method

C/17+E, B/E, D/17, A/17, Z/C

Exercise – Alternative Method

C/17+E, B/E, D/17, A/17, Z/17+E

Exercise – Alternative Method

E/17, C/17+E, B/E, D/17, A/17, Z/C

Exercise – Alternative Method

E/17, C/17+17, B/17, D/17, A/17, Z/C
Operators

- Prolog has some predefined operators (and the ability to define new ones)
- An operator is just a predicate for which a special abbreviated syntax is supported
  - Example: \(+ (2, 3)\) can also be written as \(2 + 3\)

The Predicate ‘\(' = '\)’

- The goal \(= (X, Y)\) succeeds if and only if \(X\) and \(Y\) can be unified:
  - \(?- = (\text{parent} (\text{maria}, \text{gunnar}), \text{parent} (\text{maria}, X)).\)
  - \(X = \text{gunnar}\)
    - Yes

- Since \(=\) is an operator, it can be and usually is written like this:
  - \(?- = (\text{parent} (\text{maria}, \text{gunnar}), \text{parent} (\text{maria}, X)).\)
  - \(X = \text{gunnar}\)
    - Yes

Arithmetic Operators

- Predicates \(+\), \(-\), \(*\) and \(/\) are operators too, with the usual precedence and associativity
  - \(?- X = +(1, * (2, 3)).\)
    - \(X = 1 + 2*3\)
      - Yes
  - \(?- X = 1 + 2*3.\)
    - \(X = 1 + 2*3\)
      - Yes

Prolog lets you use operator notation, and prints it out that way, but the underlying term is still \(+ (1, * (2, 3))\)

Arithmetic (‘\(\prime\)is’ gets the value)

- \(\prime\) is operator:
  - \(\prime\) is \((X, 3 + 4)\)
    - \(\prime\) is \(3 + 4\).
  - Unifies it’s first argument with the arithmetic value of its second argument.
  - Infix OK too: takes an arithmetic expression as right operand and variable as left operand
  - Variables in the expression (on right) must all be instantiated.
    - \(\prime\) is \((A, B / 10 + C)\)
      - \(A\) is \(B / 10 + C\)
    - In above, \(B\) and \(C\) needs to have been instantiated.
  - Variable on the left cannot be previously instantiated.
    - In above \(A\) cannot be instantiated (what happens if \(A\) is not a variable?)
  - Left hand side cannot be an expression since it is not evaluated -- it may be a value (and then unification is possible)

Not Evaluated

- The term is still \(+ (1, * (2, 3))\)
- It is not evaluated
- There is a way to make Prolog evaluate such terms...
### Unification impossible Example

- **Sum is Sum + Number**
- **If Sum is not instantiated, the reference to its right is undefined and the clause fails**
- **If Sum is instantiated, the clause fails because the left operand cannot have a current instantiation when it is evaluated.**

### Arithmetic Evaluation is/2

- Unifies the first argument with the value of its second argument.
  - In contrast to (=) unification predicate, which just unifies terms without evaluating them
- **Note:** left may not be a "variable" then it may unify with the value on the right.

### Trace

- **Built-in structure that displays instantiations at each step**
- **Tracing model of execution - four events:**
  - Call (beginning of attempt to satisfy goal)
  - Exit (when a goal has been satisfied)
  - Redo (when backtrack occurs)
  - Fail (when goal fails)

### List Structures

- **Other basic data structure (besides atomic propositions we have already seen): list**
- **List** is a sequence of any number of elements
- **List** is a functor of arity 2, its first component is the head and the second is the tail.
- **Elements can be atoms, atomic propositions, or other terms (including other lists)**

### Example Arithmetic

- speed(ford,100).
- speed(chevy,105).
- speed(dodge,95).
- speed(volvo,80).
- time(ford,20).
- time(chevy,21).
- time(dodge,24).
- time(volvo,24).
- distance(X,Y) :- speed(X,Speed),
  time(X,Time),
  Y is Speed * Time.

- distance(chevy, Chevy_Distance). % Query

- trace.

- distance(chevy, Chevy_Distance).
  (1) 1 Call: distance(chevy, _0)?
  (2) 2 Exit: speed(chevy, 105)
  (3) 2 Call: time(chevy, _6)?
  (4) 3 Exit: _0 is 105*21?
  (5) 3 Call: _2 is 105*21?
  (6) 4 Exit: distance(chevy, 2205)
  (2) Chevy_Distance = 2205

- distance(chevy, Chevy_Distance). % Query

- trace.

- distance(chevy, Chevy_Distance).
  (1) 1 Call: distance(chevy, _0)?
  (2) 2 Exit: speed(chevy, 105)
  (3) 2 Call: time(chevy, _6)?
  (4) 3 Exit: _0 is 105*21?
  (5) 3 Call: _2 is 105*21?
  (6) 4 Exit: distance(chevy, 2205)
  (2) Chevy_Distance = 2205

- distance(chevy, Chevy_Distance). % Query

- trace.

- distance(chevy, Chevy_Distance).
  (1) 1 Call: distance(chevy, _0)?
  (2) 2 Exit: speed(chevy, 105)
  (3) 2 Call: time(chevy, _6)?
  (4) 3 Exit: _0 is 105*21?
  (5) 3 Call: _2 is 105*21?
  (6) 4 Exit: distance(chevy, 2205)
  (2) Chevy_Distance = 2205
Same as in Scheme

nil
(a, nil)
(a, .(b, nil))
(a, .(b, .(c, .(d, .(e, nil))))))
(a,b) (note this is a pair, not a proper list)
(a, X) (this might be a list, or might not!)
(a, .(b, nil)), .(c, nil))

List Notation .() or []

- The lists is written using square brackets [].
- These are just abbreviations for the underlying term using the . Predicate
- List of length 0 is nil, denoted []

?- X = .(1, .(2, .(3, []))).
X = [1, 2, 3]
Yes
?- .(X,Y) = [1,2,3]. % head and the rest
X = 1
Y = [2, 3]
Yes

List Notation and the Tail

<table>
<thead>
<tr>
<th>List Notation</th>
<th>Term denoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>[X</td>
<td>Y]</td>
</tr>
<tr>
<td>[1</td>
<td>X]</td>
</tr>
<tr>
<td>[1,2</td>
<td>X]</td>
</tr>
<tr>
<td>[1,2</td>
<td>[3,4]]</td>
</tr>
</tbody>
</table>

- [X | Y]  
  - X is bound to first element in list, the head.
  - Y is bound to the remaining elements, called the tail.

  Useful in patterns: [1,2|X] unifies with any list that starts with 1,2 and binds X to the tail

?- [1,2|X] = [1,2,3,4,5].
X = [3, 4, 5]
Yes

The append Predicate

?- append([1,2],[3,4],Z).
Z = [1, 2, 3, 4]
Yes

- Predefined append (X, Y, Z) succeeds if and only if Z is the result of appending the list Y onto the end of the list X

?- append(X, [3,4],[1,2,3,4]).
X = [1, 2]
Yes

- append can be used with any pattern of instantiation (that is, with variables in any positions)
Implementing append()

`append([], List, List).`  
`append([Head | List1], List2, [Head | List3]) :- append(List1, List2, List3).`

- Suppose we want to join  
  » `[a, b, c]` with `[d, e]`.  
  » `[a, b, c]` has the recursive structure  
    - `[a | [b, c]]`.  
  - Then the rule says (if body is true then head is the consequence)  
    - IF `[b, c]` appends with `[d, e]` to form `[b, c, d, e]`  
    - THEN `[a | [b, c]]` appends with `[d, e]` to form `[a | [b, c, d, e]]`  
      - i.e. `[a, b, c]`  
      - `[a, b, c, d, e]`

Implementing append()

Two first parameters are the lists that are appended, the third parameters is the resulting list

- First proposition: when the empty list is appended to any other list  
  » the other list is the result.

- Second proposition:  
  » left hand side: first element of the new list (i.e. the result) is the same as the first element of the first given list (both are named `Head`).  
  » right hand side: the tail of the first given list (`List_1`) has the second given list (`List_2`) appended to form the tail of the resulting list (`List_3`).
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :- append(List_1, List_2, List_3).

trace.
append([bob, jo], [jake, darcie], Family).
(1) 1 Call: append([bob, jo], [jake, darcie], _10)?
(2) 2 Call: append([jo], [jake, darcie], _18)?
(3) 3 Call: append([], [jake, darcie], _25)?
(3) 3 Exit: append([], [jake, darcie], [jake, darcie])
(2) 2 Exit: append([jo], [jake, darcie], [jo, jake, darcie])
(1) 1 Exit: append([bob, jo], [jake, darcie], [bob, jo, jake, darcie])
Family = [bob, jo, jake, darcie]

Other Predefined List Predicates

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>member(X, Y)</td>
<td>Provable if the list Y contains the element X</td>
</tr>
<tr>
<td>select(X, Y, Z)</td>
<td>Provable if the list Y contains the element X, and Z is the same as Y but with one instance of X removed.</td>
</tr>
<tr>
<td>nth0(X, Y, Z)</td>
<td>Provable if X is an integer, Y is a list, and Z is the Xth element of Y, counting from 0.</td>
</tr>
<tr>
<td>length(X, Y)</td>
<td>Provable if X is a list of length Y.</td>
</tr>
</tbody>
</table>

● All flexible, like append
● Queries can contain variables anywhere

Using select

?- select(2, [1, 2, 3], Z).
Z = [1, 3];
No
?- select(2, Y, [1, 3]).
Y = [2, 1, 3];
Y = [1, 2, 3];
Y = [1, 3, 2];
No

?- reverse([1, 2, 3, 4], Y).
Y = [4, 3, 2, 1];
No

● Predefined reverse(X, Y) unifies Y with the reverse of the list X

Definition of reverse function:

reverse([], []).
reverse([Head | Tail], X) :-
    reverse(Tail, Y),
    append(Result, [Head], X).

This step is wrong: we substituted X for Y, but there is already a different X elsewhere in the goal list.

reverse([[], []], X) :- reverse([], []).
This step is wrong: we substituted $X$ for $Y$, but there is already a different $X$ elsewhere in the goal list.

```
reverse([], []).  
reverse([Head|Tail], X) :-  
    reverse(Tail, Y),  
    append(Y, [Head], X).
```

```
reverse([], []).  
reverse([2], Y)  
    append(Y2, [2], X2)  
    append(X2, [1], X1)  
    reverse([], Y2)  
    append([], [2], X2),  
    append(X2, [1], X1)  
    append([2], [1], X1)  
    solve [
```

```
Advantages:

- Prolog programs based on logic, so likely to be more logically organized and written
- Processing is naturally parallel, so Prolog interpreters can take advantage of multi-processor machines
- Programs are concise, so development time is decreased – good for prototyping

Deficiencies of Prolog

- Resolution order control
- The closed-world assumption
- The negation problem
- Intrinsic limitations

SWI-Prolog

```
?- set_prolog_flag(history, 50).
Yes
27 ?- h.  % shows history of commands
  2   eats(Person1, Food1).  
  3   eats(Person1, Food), eats(Person2, Food).  
  4   eats(corey, fish).  
  ?- !!.  % Repeats last query
```