

A First Look At Prolog

Outline

- Terms
- Using a Prolog language system
- Rules
- The two faces of Prolog
- Operators
- Lists
- Negation and failure
- What Prolog is good for

Terms

- Everything in Prolog is built from *terms*:
 - Prolog programs
 - The data manipulated by Prolog programs
- Three kinds of terms:
 - Constants: integers, real numbers, atoms
 - Variables
 - Compound terms

Constants

- Integer constants: **123**
- Real constants: **1.23**
- Atoms:
 - A lowercase letter followed by any number of additional letters, digits or underscores: **fred**
 - A sequence of non-alphanumeric characters:
***, ., =, @#\$**
 - Plus a few special atoms: **[]**

Atoms Are Not Variables

- An atom can look like an ML or Java variable:
 - **i, size, length**
- But an atom is not a variable; it is not bound to anything, never equal to anything else
- Think of atoms as being more like string constants: **"i", "size", "length"**

Variables

- Any name beginning with an uppercase letter or an underscore, followed by any number of additional letters, digits or underscores: **X**, **Child**, **Fred**, **_**, **_123**
- Most of the variables you write will start with an uppercase letter
- Those starting with an underscore, including **_**, get special treatment

Compound Terms

- An atom followed by a parenthesized, comma-separated list of one or more terms:
 $x(y, z), +(1, 2), .(1, []),$
`parent(adam, seth), x(Y, x(Y, Z))`
- A compound term can look like an ML function call: **$f(x, y)$**
- Again, this is misleading
- Think of them as structured data

Terms

$\langle term \rangle ::= \langle constant \rangle \mid \langle variable \rangle \mid \langle compound-term \rangle$
 $\langle constant \rangle ::= \langle integer \rangle \mid \langle real\ number \rangle \mid \langle atom \rangle$
 $\langle compound-term \rangle ::= \langle atom \rangle (\langle termlist \rangle)$
 $\langle termlist \rangle ::= \langle term \rangle \mid \langle term \rangle , \langle termlist \rangle$

- 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

Unification

- Pattern-matching using Prolog terms
- Two terms *unify* if there is some way of binding their variables that makes them identical
- For instance, **parent(adam, Child)** and **parent(adam, seth)** unify by binding the variable **Child** to the atom **seth**
- More details later: Chapter 20

The Prolog Database

- A Prolog language system maintains a collection of facts and rules of inference
- It is like an internal database that changes as the Prolog language system runs
- 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

Example

```
parent(kim,holly) .  
parent(margaret,kim) .  
parent(margaret,kent) .  
parent(esther,margaret) .  
parent(herbert,margaret) .  
parent(herbert,jean) .
```

- A Prolog program of six facts
- Defining a *predicate* **parent** of *arity* 2
- We would naturally interpret these as facts about families: Kim is the parent of Holly and so on

Outline

- Terms
- Using a Prolog language system
- Rules
- The two faces of Prolog
- Operators
- Lists
- Negation and failure
- What Prolog is good for

SWI-Prolog

```
Welcome to SWI-Prolog ...
```

```
For help, use ?- help(Topic) . or ?- apropos(Word) .
```

```
?-
```

- Prompting for a query with `?-`
- Normally interactive: get query, print result, repeat

The `consult` Predicate

```
?- consult(relations).  
% relations compiled 0.00 sec, 852 bytes  
true.  
  
?-
```

- Predefined predicate to read a program from a file into the database
- File **relations** (or **relations.pl**) contains our **parent** facts

Simple Queries

```
?- parent(margaret,kent) .  
true.  
  
?- parent(fred,pebbles) .  
false.  
  
?-
```

- A query asks the language system to prove something
- Some turn out to be **true**, some **false**
- (Some queries, like **consult**, are executed only for their side-effects)

Final Period

```
?- parent(margaret,kent)
|
true.

?-
```

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

Queries With Variables

```
?- parent(P,jean) .  
P = herbert.  
  
?- parent(P,esther) .  
false.
```

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

Flexibility

- Normally, variables can appear in any or all positions in a query:
 - `parent (Parent, jean)`
 - `parent (esther, Child)`
 - `parent (Parent, Child)`
 - `parent (Person, Person)`

Multiple Solutions

```
?- parent(Parent,Child) .  
Parent = kim,  
Child = holly .
```

- When the system finds a solution, it prints the binding it found
- If it could continue to search for additional solutions, it then prompts for input
- Hitting Enter makes it stop searching and print the final period...

Multiple Solutions

- ... entering a semicolon makes it continue the search
- As often as you do this, it will try to find another solution
- In this case, there is one for every fact in the database

```
?- parent(Parent,Child) .  
Parent = kim,  
Child = holly ;  
Parent = margaret,  
Child = kim ;  
Parent = margaret,  
Child = kent ;  
Parent = esther,  
Child = margaret ;  
Parent = herbert,  
Child = margaret ;  
Parent = herbert,  
Child = jean.
```

Conjunctions

```
?- parent(margaret,X) , parent(X,holly) .  
X = kim .
```

- A conjunctive query has a list of query terms separated by commas
- The Prolog system tries prove them all (using a single set of bindings)

```
?- parent(Parent,kim) , parent(Grandparent,Parent) .  
Parent = margaret,  
Grandparent = esther ;  
Parent = margaret,  
Grandparent = herbert ;  
false.
```

```
?- parent(esther,Child) ,  
|   parent(Child,Grandchild) ,  
|   parent(Grandchild,GreatGrandchild) .  
Child = margaret,  
Grandchild = kim,  
GreatGrandchild = holly .
```

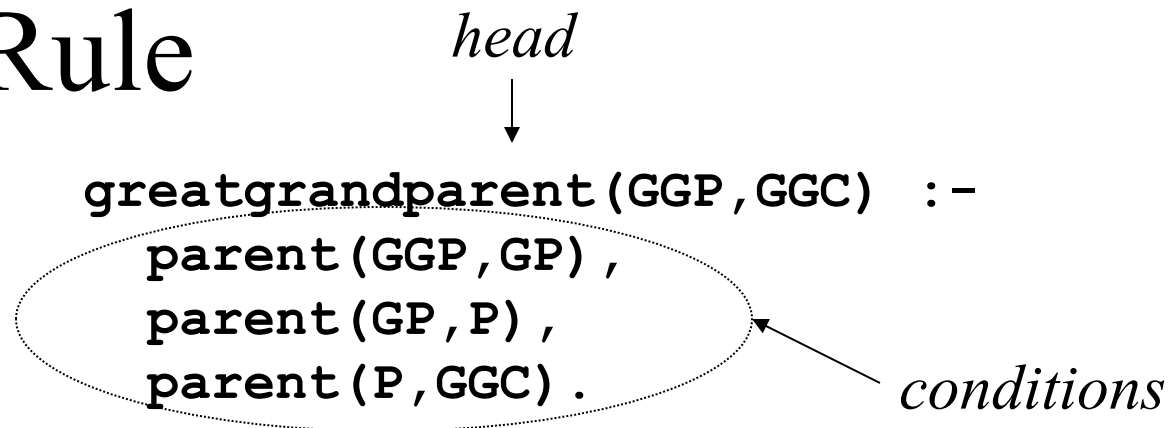
Outline

- Terms
- Using a Prolog language system
- **Rules**
- The two faces of Prolog
- Operators
- Lists
- Negation and failure
- What Prolog is good for

The Need For Rules

- Previous example had a lengthy query for great-grandchildren of Esther
- It would be nicer to query directly:
`greatgrandparent (esther , GGC)`
- But we do not want to add separate facts of that form to the database
- The relation should follow from the **parent** relation already defined

A Rule



- A rule says how to prove something: to prove the head, prove the conditions
- To prove `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 Program With The Rule

```
parent(kim,holly) .  
parent(margaret,kim) .  
parent(margaret,kent) .  
parent(esther,margaret) .  
parent(herbert,margaret) .  
parent(herbert,jean) .  
greatgrandparent(GGP,GGC) :-  
    parent(GGP,GP) , parent(GP,P) , parent(P,GGC) .
```

- A program consists of a list of *clauses*
- A clause is either a fact or a rule, and ends with a period

Example

```
?- greatgrandparent (esther, GreatGrandchild) .  
GreatGrandchild = holly .
```

- This shows the initial query and final result
- Internally, there are intermediate *goals*:
 - The first goal is the initial query
 - The next is what remains to be proved after transforming the first goal using one of the clauses (in this case, the greatgrandparent rule)
 - And so on, until nothing remains to be proved

1. `parent(kim,holly) .`
 2. `parent(margaret,kim) .`
 3. `parent(margaret,kent) .`
 4. `parent(esther,margaret) .`
 5. `parent(herbert,margaret) .`
 6. `parent(herbert,jean) .`
 7. `greatgrandparent(GGP,GGC) :-`
 `parent(GGP,GP) , parent(GP,P) , parent(P,GGC) .`
- We will see more about Prolog's model of execution in Chapter 20*

`greatgrandparent(esther,GreatGrandchild)`

↓ Clause 7, binding `GGP` to `esther` and `GGC` to `GreatGrandChild`

`parent(esther,GP) , parent(GP,P) , parent(P,GreatGrandchild)`

↓ Clause 4, binding `GP` to `margaret`

`parent(margaret,P) , parent(P,GreatGrandchild)`

↓ Clause 2, binding `P` to `kim`

`parent(kim,GreatGrandchild)`

↓ Clause 1, binding `GreatGrandchild` to `holly`

Rules Using Other Rules

```
grandparent(GP,GC) :-  
    parent(GP,P) , parent(P,GC) .
```

```
greatgrandparent(GGP,GGC) :-  
    grandparent(GGP,P) , parent(P,GGC) .
```

- Same relation, defined indirectly
- Note that both clauses use a variable **P**
- The scope of the definition of a variable is the clause that contains it

Recursive Rules

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

- **X** is an ancestor of **Y** if:
 - 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 you give them, so put base-case rules and facts first

```
?- ancestor(jean,jean) .  
false.
```

```
?- ancestor(kim,holly) .  
true .
```

```
?- ancestor(A,holly) .  
A = kim ;  
A = margaret ;  
A = esther ;  
A = herbert ;  
false.
```

Core Syntax Of Prolog

- You have seen the complete core syntax:

$$\begin{aligned} \langle clause \rangle & ::= \langle fact \rangle \mid \langle rule \rangle \\ \langle fact \rangle & ::= \langle term \rangle . \\ \langle rule \rangle & ::= \langle term \rangle :- \langle termlist \rangle . \\ \langle termlist \rangle & ::= \langle term \rangle \mid \langle term \rangle , \langle termlist \rangle \end{aligned}$$

- There is not much more syntax for Prolog than this: it is a very simple language
- Syntactically, that is!

Outline

- Terms
- Using a Prolog language system
- Rules
- The two faces of Prolog
- Operators
- Lists
- Negation and failure
- What Prolog is good for

The Procedural Side

```
greatgrandparent (GGP,GGC) :-  
    parent (GGP,GP) , parent (GP,P) , parent (P,GGC) .
```

- A rule says how to prove something:
 - To prove `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 Prolog program specifies proof procedures for queries

The Declarative Side

- A rule is a logical assertion:
 - For all bindings of `GGP`, `GP`, `P`, and `GGC`, if `parent(GGP, GP)` and `parent(GP, P)` and `parent(P, GGC)`, then `greatgrandparent(GGP, GGC)`
- Just a formula – it doesn't say how to *do* anything – it just makes an assertion:

$$\forall GGP, GP, P, GGC . \text{parent}(GGP, GP) \wedge \text{parent}(GP, P) \wedge \text{parent}(P, GGC) \\ \Rightarrow \text{greatgrandparent}(GGP, GGC)$$

Declarative Languages

- Each piece of the program corresponds to a simple mathematical abstraction
 - Prolog clauses – formulas in first-order logic
 - ML fun definitions – functions
- Many people use *declarative* as the opposite of *imperative*, including both logic languages and functional languages

Declarative Advantages

- Imperative languages are doomed to subtle side-effects and interdependencies
- Simpler declarative semantics makes it easier to develop and maintain correct programs
- Higher-level, more like *automatic programming*: describe the problem and have the computer write the program

Prolog Has Both Aspects

- Partly declarative
 - A Prolog program has logical content
- Partly procedural
 - A Prolog program has procedural concerns: clause ordering, condition ordering, side-effecting predicates, etc.
- It is important to be aware of both

Outline

- Terms
- Using a Prolog language system
- Rules
- The two faces of Prolog
- **Operators**
- Lists
- Negation and failure
- What Prolog is good for

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

The = Predicate

- The goal $\text{=}(\mathbf{X}, \mathbf{Y})$ succeeds if and only if \mathbf{X} and \mathbf{Y} can be unified:

```
?- = (parent (adam, seth) , parent (adam, X) ) .  
X = seth.
```

- Since = is an operator, it can be and usually is written like this:

```
?- parent (adam, seth) = parent (adam, X) .  
X = seth.
```

Arithmetic Operators

- Predicates $+$, $-$, $*$ and $/$ are operators too, with the usual precedence and associativity

```
?- X = +(1, *(2,3)).  
X = 1+2*3.
```

```
?- X = 1+2*3.  
X = 1+2*3.
```

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

Not Evaluated

```
?- +(X,Y) = 1+2*3.  
X = 1,  
Y = 2*3.  
  
?- 7 = 1+2*3.  
false.
```

- The term is still $+(1, *(2, 3))$
- It is not evaluated
- There is a way to make Prolog evaluate such terms, but we won't need it yet

Outline

- Terms
- Using a Prolog language system
- Rules
- The two faces of Prolog
- Operators
- **Lists**
- Negation and failure
- What Prolog is good for

Lists in Prolog

- A bit like ML lists
- The atom `[]` represents the empty list
- A predicate `.` corresponds to ML's `::` operator

ML expression	Prolog term
<code>[]</code>	<code>[]</code>
<code>1 :: []</code>	<code>. (1, [])</code>
<code>1 :: 2 :: 3 :: []</code>	<code>. (1, . (2, . (3, [])))</code>
No equivalent.	<code>. (1, . (parent(X,Y), []))</code>

List Notation

List notation	Term denoted
<code>[]</code>	<code>[]</code>
<code>[1]</code>	<code>. (1, [])</code>
<code>[1, 2, 3]</code>	<code>. (1, . (2, . (3, [])))</code>
<code>[1, parent(X, Y)]</code>	<code>. (1, . (parent(X, Y), []))</code>

- ML-style notation for lists
- These are just abbreviations for the underlying term using the `.` Predicate
- Prolog usually displays lists in this notation

Example

```
?- X = . (1, . (2, . (3, []))) .  
x = [1, 2, 3] .
```

```
?- . (X, Y) = [1, 2, 3] .  
x = 1,  
y = [2, 3] .
```

List Notation With Tail

List notation	Term denoted
[1 X]	. (1, X)
[1, 2 X]	. (1, . (2, X))
[1, 2 [3, 4]]	same as [1, 2, 3, 4]

- Last in a list can be the symbol **|** followed by a final term for the tail of the list
- 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].
```


The **append** Predicate

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

- 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**

Not Just A Function

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

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

Not Just A Function

```
?- append(X,Y,[1,2,3]).  
X = [],  
Y = [1, 2, 3] ;  
X = [1],  
Y = [2, 3] ;  
X = [1, 2],  
Y = [3] ;  
X = [1, 2, 3],  
Y = [] ;  
false.
```

An Implementation

```
append([], B, B) .  
append([Head|TailA], B, [Head|TailC]) :-  
    append(TailA, B, TailC) .
```

Other Predefined List Predicates

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

- All flexible, like **append**
- Queries can contain variables anywhere

Using **select**

```
?- select(2,[1,2,3],Z) .  
Z = [1, 3] ;  
false.
```

```
?- select(2,Y,[1,3]) .  
Y = [2, 1, 3] ;  
Y = [1, 2, 3] ;  
Y = [1, 3, 2] ;  
false.
```

The **reverse** Predicate

```
?- reverse([1,2,3,4],Y) .  
Y = [4, 3, 2, 1] .
```

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

An Implementation

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

- Not an efficient way to reverse
- We'll see why, and a more efficient solution, in Chapter 21

Non-Terminating Queries

```
?- reverse(X, [1,2,3,4]).
```

```
X = [4, 3, 2, 1] ;
```

```
^CAction (h for help) ? abort
```

```
% Execution Aborted
```

```
?-
```

- Asking for another solution caused an infinite loop
- Hit Control-C to stop it, then *a* for abort
- **reverse** cannot be used as flexibly as **append**

Flexible and Inflexible

- Ideally, predicates should all be flexible like **append**
- They are more declarative, with fewer procedural quirks to consider
- But inflexible implementations are sometimes used, for efficiency or simplicity
- Another example is **sort...**

Example

```
?- sort([2,3,1,4],X).
```

```
X = [1, 2, 3, 4].
```

```
?- sort(X,[1,2,3,4]).
```

```
ERROR: Arguments are not sufficiently instantiated
```

- A fully flexible **sort** would also be able to unsort—find all permutations
- But it would not be as efficient for the more common task

The Anonymous Variable

- The variable `_` is an anonymous variable
- Every occurrence is bound independently of every other occurrence
- In effect, much like ML's `_`: it matches any term without introducing bindings

Example

`tailof([_|A],A) .`

- This **`tailof(X,Y)`** succeeds when **`X`** is a non-empty list and **`Y`** is the tail of that list
- Don't use this, even though it works:

`tailof([Head|A],A) .`

Dire Warning

```
append([], B, B) .  
append([Head|TailA], B, [Head|TailC]) :-  
    append(TailA, B, TailC) .
```

- Don't ignore warning message about singleton variables
- As in ML, it is bad style to introduce a variable you never use
- More importantly: *if you misspell a variable name, this is the only warning you will see*

Outline

- Terms
- Using a Prolog language system
- Rules
- The two faces of Prolog
- Operators
- Lists
- **Negation and failure**
- What Prolog is good for

The **not** Predicate

```
?- member(1,[1,2,3]).  
true .  
  
?- not(member(4,[1,2,3])).  
false.▲
```

- For simple applications, it often works quite a bit logical negation
- But it has an important procedural side...

Negation As Failure

- To prove **not (X)** , Prolog attempts to prove **X**
- **not (X)** succeeds if **X** fails
- The two faces again:
 - Declarative: **not (X)** = $\neg X$
 - Procedural: **not (X)** succeeds if **X** fails, fails if **X** succeeds, and runs forever if **X** runs forever

Example

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

```
?- sibling(kim,kent) .  
true .
```

```
?- sibling(kim,kim) .  
false.
```

```
?- sibling(X,Y) .  
false.
```

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

```
?- sibling(X,Y) .  
X = kim,  
Y = kent ;  
X = kent,  
Y = kim ;  
X = margaret,  
Y = jean ;  
X = jean,  
Y = margaret ;  
false.
```

Outline

- Terms
- Using a Prolog language system
- Rules
- The two faces of Prolog
- Operators
- Lists
- Negation and failure
- What Prolog is good for

A Classic Riddle

- A man travels with wolf, goat and cabbage
- Wants to cross a river from west to east
- A rowboat is available, but only large enough for the man plus one possession
- Wolf eats goat if left alone together
- Goat eats cabbage if left alone together
- How can the man cross without loss?

Configurations

- Represent a configuration of this system as a list showing which bank each thing is on in this order: man, wolf, goat, cabbage
- Initial configuration: **[w, w, w, w]**
- If man crosses with wolf, new state is **[e, e, w, w]** – but then goat eats cabbage, so we can't go through that state
- Desired final state: **[e, e, e, e]**

Moves

- In each move, man crosses with at most one of his possessions
- We will represent these four moves with four atoms: **wolf**, **goat**, **cabbage**, **nothing**
- (Here, **nothing** indicates that the man crosses alone in the boat)

Moves Transform Configurations

- Each move transforms one configuration to another
- In Prolog, we will write this as a predicate:
move (Config, Move, NextConfig)
 - **Config** is a configuration (like **[w, w, w, w]**)
 - **Move** is a move (like **wolf**)
 - **NextConfig** is the resulting configuration (in this case, **[e, e, w, w]**)

The **move** Predicate

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

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

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

```
move([X,Wolf,X,Cabbage],goat,[Y,Wolf,Y,Cabbage]) :-  
    change(X,Y) .
```

```
move([X,Wolf,Goat,X],cabbage,[Y,Wolf,Goat,Y]) :-  
    change(X,Y) .
```

```
move([X,Wolf,Goat,C],nothing,[Y,Wolf,Goat,C]) :-  
    change(X,Y) .
```


Safe Configurations

- A configuration is safe if
 - At least one of the goat or the wolf is on the same side as the man, and
 - At least one of the goat or the cabbage is on the same side as the man

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

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

Solutions

- A solution is a starting configuration and a list of moves that takes you to **[e, e, e, e]**, where all the intermediate configurations are safe

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

Prolog Finds A Solution

```
?- length(X,7), solution([w,w,w,w],X).  
X = [goat, nothing, wolf, goat, cabbage, nothing, goat] .
```

- Note: without the **length(X,7)** restriction, Prolog would not find a solution
- It gets lost looking at possible solutions like **[goat,goat,goat,goat,goat...]**
- More about this in Chapter 20

What Prolog Is Good For

- The program specified a problem logically
- It did not say how to search for a solution to the problem – Prolog took it from there
- That's one kind of problem Prolog is especially good for
- More examples to come in Chapter 22