Introduction to Logic Programming in Prolog
Aug 2, 2017
Outline

Programming paradigms

Logic programming basics
   Introduction to Prolog
   Predicates, queries and rules

Understanding the query engine
   Goal search and unification
   Structuring recursive rules

Complex terms and lists

Cuts and negation
Programming paradigms
What is a programming paradigm?

**Paradigm**: A conceptual model underlying the theories and practice of a *scientific subject*

*scientific subject* = *programming*

**Programming paradigm**: A conceptual model underlying the theories and practice of *programming*
### Imperative paradigm

#### Imperative model

<table>
<thead>
<tr>
<th>data</th>
<th>set of state variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>transformation of state</td>
</tr>
</tbody>
</table>

Type State = [(Name, Val)]

State → State

Needs two sub-languages:

- **expressions** to describe values to store in variables (Expr)
- **statements** to describe state changes and control flow (Stmt)

Semantic functions:

- \( \text{semE} :: \text{Expr} \rightarrow \text{State} \rightarrow \text{Val} \)
- \( \text{semS} :: \text{Stmt} \rightarrow \text{State} \rightarrow \text{State} \)
Object-oriented paradigm

An extension/refinement of the imperative paradigm

Object-oriented model

<table>
<thead>
<tr>
<th>data</th>
<th>set of objects with state</th>
</tr>
</thead>
<tbody>
<tr>
<td>type Object</td>
<td>(State, [Method])</td>
</tr>
<tr>
<td>type Method</td>
<td>(Name, State -&gt; State)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>computation</th>
<th>evolution of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Object] -&gt; [Object]</td>
<td></td>
</tr>
</tbody>
</table>

Needs **expression** and **statement** sub-languages, but also extended statements with:
- constructs to **create objects** and **invoke methods**

New statement semantic function:
- \( \text{semS} :: \text{Stmt} \to [\text{Object}] \to [\text{Object}] \)
Functional paradigm

**Functional model**

<table>
<thead>
<tr>
<th>data</th>
<th>structured values</th>
<th>data Val = ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>functions over values</td>
<td>Val -&gt; Val</td>
</tr>
</tbody>
</table>

Generally just one language (e.g. lambda calculus):

- **expressions** describe functions and values (**Expr**)

Semantic functions:

- **sem :: Expr -> Val**
Logic paradigm

**Logical model**

<table>
<thead>
<tr>
<th>data</th>
<th>set of values and relations</th>
<th>type Known = [(Val,...,Val)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>query over relations</td>
<td>type Query = Known -&gt; Known</td>
</tr>
</tbody>
</table>

Generally just one language:

- **relations** describe both knowledge and queries (**Rel**)

Semantic functions:

- **sem :: Rel -> Query**
## Comparison of programming paradigms

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>View of computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>imperative</td>
<td>sequence of state transformations</td>
</tr>
<tr>
<td>object-oriented</td>
<td>simulation of interacting objects</td>
</tr>
<tr>
<td>functional</td>
<td>function mapping input to output</td>
</tr>
<tr>
<td>logic</td>
<td>queries over logical relations</td>
</tr>
</tbody>
</table>
Outline

Programming paradigms

Logic programming basics
  Introduction to Prolog
  Predicates, queries and rules

Understanding the query engine
  Goal search and unification
  Structuring recursive rules

Complex terms and lists

Cuts and negation
What is Prolog?

- an **untyped logic** programming language
- programs are **rules** that define **relations** on values
- run a program by formulating a **goal** or **query**
- result of a program: a true/false answer and a **binding of free variables**
Logic: a tool for reasoning

**Syllogism** (logical argument) — Aristotle, 350 BCE

*Every human is mortal.*

*Socrates is human.*

*Therefore, Socrates is mortal.*

**First-order logic** — Gottlob Frege, 1879 *Begriffsschrift*

\[
\forall x. \text{Human}(x) \rightarrow \text{Mortal}(x)
\]

\[
\text{Human}(\text{Socrates})
\]

\[\therefore \text{Mortal}(\text{Socrates})\]
Logic and programming

**Rule**  \( \forall x. \text{Human}(x) \rightarrow \text{Mortal}(x) \)

**Fact**  \( \text{Human}(\text{Socrates}) \)

**Goal/Query**  \( \therefore \text{Mortal}(\text{Socrates}) \)

---

**Prolog program**

```
mortal(X) :- human(X).
human(Socrates).
```

**Prolog query (interactive)**

```
?- mortal(Socrates).
true.
```
Outline

Programming paradigms

Logic programming basics
  Introduction to Prolog
  Predicates, queries and rules

Understanding the query engine
  Goal search and unification
  Structuring recursive rules

Complex terms and lists

Cuts and negation
## SWI-Prolog logistics

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[myfile].</td>
<td>load definitions from “myfile.pl”</td>
</tr>
<tr>
<td>listing(P).</td>
<td>lists facts and rules related to predicate $P$</td>
</tr>
<tr>
<td>trace.</td>
<td>turn on tracing</td>
</tr>
<tr>
<td>nodebug.</td>
<td>turn off tracing</td>
</tr>
<tr>
<td>help.</td>
<td>open help window (requires X11 on Mac)</td>
</tr>
<tr>
<td>halt.</td>
<td>quit</td>
</tr>
</tbody>
</table>

GNU-Prolog uses the same commands — except help!
Atoms

An **atom** is just a primitive value

- string of characters, numbers, underscores starting with a **lowercase letter**:
  - `hello`, `socrates`, `uP_aNd_4tOm`

- any single quoted string of characters:
  - `‘Hello world!’`, `‘Socrates’`

- numeric literals: `123`, `–345`

- empty lists: `[]`
Variables

A variable can be used in rules and queries

- string of characters, numbers, underscores starting with an uppercase letter or an underscore
  - X, SomeHuman, _g_123, This_Human

- special variable: _ (just an underscore)
  - unifies with anything — “don’t care”
Predicates

Basic entity in Prolog is a predicate \( \equiv \) relation \( \equiv \) set

**Unary predicate**
- hobbit(bilbo).
- hobbit(frodo).
- hobbit(sam).

\[ \text{hobbit} = \{ \text{bilbo, frodo, sam} \} \]

**Binary predicate**
- likes(bilbo, frodo).
- likes(frodo, bilbo).
- likes(sam, frodo).
- likes(frodo, ring).

\[ \text{likes} = \{(\text{bilbo, frodo}), (\text{frodo, bilbo}), (\text{sam, frodo}), (\text{frodo, ring})\} \]
Simple goals and queries

Predicates are:

• **defined** in a file  
• **queried** in the REPL

Response to a query is a **true/false** answer  
(or **yes/no**)  
when true, provides a **binding** for each variable in the query

**Is sam a hobbit?**
?- hobbit(sam).
true.

**Is gimli a hobbit?**
?- hobbit(gimli).
false.

**Who is a hobbit?**
?- hobbit(X).
X = bilbo ;
X = frodo ;
X = sam .

**Type ; after each response to search for another**
Querying relations

You can query **any argument** of a predicate

- this is fundamentally different from passing arguments to functions!

**Definition**

\[
\text{likes(bilbo, frodo).} \\
\text{likes(frodo, bilbo).} \\
\text{likes(sam, frodo).} \\
\text{likes(frodo, ring).}
\]

?- \text{likes(frodo, Y).} \\
Y = \text{bilbo} ; \\
Y = \text{ring} .

?- \text{likes(X, Y).} \\
X = \text{bilbo,} \\
Y = \text{frodo} ; \\
X = \text{frodo,} \\
Y = \text{bilbo} ; \\
X = \text{sam,} \\
Y = \text{frodo} ; \\
X = \text{frodo,} \\
Y = \text{ring} .

?- \text{likes(X, frodo).} \\
X = \text{bilbo} ; \\
X = \text{sam} .
Overloading predicates

Predicates with the **same name** but **different arities** are **different predicates**!

```prolog
hobbit/1
hobbit(bilbo).
hobbit(frodo).
hobbit(sam).

?- hobbit(X).
   X = bilbo ;
   X = frodo ;
   X = sam .

hobbit/2
hobbit(bilbo, rivendell).
hobbit(frodo, hobbiton).
hobbit(sam, hobbiton).
hobbit(merry, buckland).
hobbit(pippin, tookland).

?- hobbit(X,_).
   ... 
   X = merry ;
   X = pippin .
```

hobbits.pl
Conjunction

Comma (,) denotes **logical and** of two predicates

**Do sam and frodo like each other?**
?- likes(sam,frodo), likes(frodo,sam).
true.

**Do merry and pippin live in the same place?**
?- hobbit(merry,X), hobbit(pippin,X).
false.

**Do any hobbits live in the same place?**
?- hobbit(H1,X), hobbit(H2,X), H1 \(!=\) H2.
H1 = frodo, X = hobbiton, H2 = sam.

likes(frodo, sam).
likes(sam, frodo).
likes(frodo, ring).
hobbit(frodo, hobbiton).
hobbit(sam, hobbiton).
hobbit(merry, buckland).
hobbit(pippin, tookland).

H1 and H2 must be different!
Rules

**Rule:** head :- body

The head is true if the body is true

**Examples**

likes(X,beer) :- hobbit(X,\_).
likes(X,boats) :- hobbit(X,buckland).

danger(X) :- likes(X,ring).
danger(X) :- likes(X,boats), likes(X,beer).

Note that **disjunction** is described by multiple rules
Prolog

Penguins are black and white. Some old TV shows are black and white. Therefore, some penguins are old TV shows.

Logic: another thing that penguins aren’t very good at.
Outline

Programming paradigms

Logic programming basics
  Introduction to Prolog
  Predicates, queries and rules

Understanding the query engine
  Goal search and unification
  Structuring recursive rules

Complex terms and lists

Cuts and negation
How does Prolog solve queries?

Basic algorithm for solving a (sub)goal
1. linearly search data base for candidate facts/rules
2. attempt to unify candidate with goal
   if unification is successful:
     • if a fact — we’re done with this goal!
     • if a rule — add body of rule as new subgoal
   if unification is unsuccessful: keep searching
3. backtrack if we reach the end of the database
1. linearly search the database for candidate facts/rules

What is a candidate fact/rule?

- **fact**: predicate matches the goal
- **rule**: predicate of its head matches the goal

Example goal: `likes(merry,Y)`

**Candidates**
- `likes(sam,frodo).
- `likes(merry,pippin).
- `likes(X,beer) :- hobbit(X).

**Not candidates**
- `hobbit(merry,buckland).
- `danger(X) :- likes(X,ring).
- `likes(merry,pippin,mushrooms).`
2. attempt to unify candidate with goal

**Unification**
Find an **assignment of variables** that makes its arguments **syntactically equal**
Prolog: \( A = B \) means attempt to **unify** \( A \) and \( B \)

**Candidates**

?- likes(merry,Y) = likes(sam,frodo).
false.

?- likes(merry,Y) = likes(merry,pippin).
\( Y = \text{pippin} \).

?- likes(merry,Y) = likes(X,beer).
\( X = \text{merry} ; Y = \text{beer} \).

2a. if **fail**, try next candidate
2b. if **success**, add new subgoal(s)
Tracking subgoals

Deriving solutions through rules

1. maintain a list of goals that need to be solved
   • when this list is empty we’re finished!
2. if current goal unifies with a rule head, add body as subgoals to list
3. after unification, substitute variables in all goals in the list!

Database

Sequence of goals for \( \text{lt}(\text{one}, \text{four}) \)

\[
\begin{align*}
1 & \quad \text{lt}(\text{one}, \text{two}). \\
2 & \quad \text{lt}(\text{two}, \text{three}). \\
3 & \quad \text{lt}(\text{three}, \text{four}). \\
4 & \quad \text{lt}(X, Z) :- \text{lt}(X, Y), \text{lt}(Y, Z).
\end{align*}
\]

1: \( Y1 = \text{two} \)  \\
2: \( Y2 = \text{three} \)  \\
3: true

\[
\begin{align*}
4: & \quad X = \text{one}, Z = \text{four} \\
1: & \quad Y1 = \text{two} \\
4: & \quad X = \text{two}, Z = \text{four} \\
2: & \quad Y2 = \text{three} \\
3: & \quad \text{true} \quad \text{done!}
\end{align*}
\]
3. Backtracking

For each subgoal, Prolog maintains:

- the search state (goals + assignments) before it was produced
- a pointer to the rule that produced it

When a subgoal fails:

- restore the previous state
- resume search for previous goal from the pointer

When the initial goal fails: return false
Outline

Programming paradigms

Logic programming basics
  Introduction to Prolog
  Predicates, queries and rules

Understanding the query engine
  Goal search and unification
    Structuring recursive rules

Complex terms and lists

Cuts and negation
Potential for infinite search

Why care about how goal searches work?

One reason: so we can write recursive rules that don’t loop forever!

**Contra-example: symmetry**
likes(frodo,sam).
likes(merry,pippin).
likes(frodo,bilbo).
likes(X,Y) :- likes(Y,X).

?- likes(bilbo,merry).
ERROR: Out of local stack

**Contra-example: transitivity**
lt(one,two).
lt(two,three).
lt(three,four).
lt(X,Z) :- lt(X,Y), lt(Y,Z).

?- lt(three,one).
ERROR: Out of local stack
Strategies for writing recursive rules

How to avoid infinite search
1. always list **non-recursive cases first**
2. use “helper” predicates to **enforce progress** during search

Example: symmetry
\[\text{likesP(frodo, sam).} \]
\[\text{likesP(merry, pippin).} \]
\[\text{likesP(frodo, bilbo).} \]
\[\text{likes(X, Y) :- likesP(X, Y).} \]
\[\text{likes(X, Y) :- likesP(Y, X).} \]

?- \text{likes(bilbo, merry).}
false.

Example: transitivity
\[\text{ltP(one, two).} \]
\[\text{ltP(two, three).} \]
\[\text{ltP(three, four).} \]
\[\text{lt(X, Y) :- ltP(X, Y).} \]
\[\text{lt(X, Z) :- ltP(X, Y), lt(Y, Z).} \]

?- \text{lt(three, one).}
false.
Outline

Programming paradigms

Logic programming basics
  Introduction to Prolog
  Predicates, queries and rules

Understanding the query engine
  Goal search and unification
  Structuring recursive rules

Complex terms and lists

Cuts and negation
Representing structured data

Can represent structured data by **nested predicates**

**Example database**

rides(gandalf, horse(white)).
rides(sam, donkey(grey)).
rides(frodo, pony(grey)).
rides(strider, horse(black)).

?- rides(gandalf, X).
X = horse(white) .

?- rides(X, horse(Y)).
X = gandalf, Y = white ;
X = strider, Y = black .

Variables *cannot* be used for predicates:
?- rides(X, Y(grey)). ← illegal!
Relationship to Haskell data types

**Haskell data type**

```haskell
data Exp = Lit Int | Neg Exp | Add Exp Exp | Mul Exp Exp
```

```
Add (Neg (Lit 3))
(Mul (Lit 4) (Lit 5))
```

- build values w/data constructors
- data types statically define valid combinations

**Prolog predicate**

```prolog
exp(N) :- number(N).
exp(neg(E)) :- exp(E).
exp(add(L,R)) :- exp(L), exp(R).
exp(mul(L,R)) :- exp(L), exp(R).
```

```
add(neg(3),mul(4,5))
```

- build values w/predicates
- use rules to dynamically identify or enumerate valid combinators
Equality

Different forms of equality between terms

1. unification  \`=`  and  \`\}=`  
2. equivalence  \`==`  and  \`\\==`  
3. evaluation  \`\=:=`  and  \`\=\\=`
Unification equality

Two terms are *equal* when they can be *instantiated* so that they *become identical*.

?- 3=3.  true.
?- X=3.  X = 3.
?- X=Y.  X = Y.

?- likes(X,red)=likes(john,Y).
    X = john,
    Y = red.

?- car(red)=car(X).
    X = red.

?- 3=4.  false.
?- 3+1=4. false.
Unification

A unifier for two terms $T$ and $T'$ is a substitution for variables $\sigma$, such that:

$$\sigma(T) = \sigma(T')$$

?- 3=3.
true.

$\sigma = \{\}$
$\sigma(3) = 3 = \sigma(3)$

?- X=3.
X = 3.

$\sigma = \{X \rightarrow 3\}$
$\sigma(X) = 3 = \sigma(3)$

?- X=Y.
X = Y.

$\sigma = \{X \rightarrow Y\}$
$\sigma(X) = Y = \sigma(Y)$

?- likes(X,red)=likes(john,Y).
X = john,
Y = red.

$\sigma = \{X \rightarrow john, Y \rightarrow red\}$
$\sigma(\text{likes}(X,red)) = \text{likes}(john,red)
\quad = \sigma(\text{likes}(john,Y))$

?- car(red)=car(X).
X = red.

$\sigma = \{X \rightarrow red\}$
$\sigma(\text{car}(red)) = \sigma(\text{car}(X))$
Equivalence

Two terms are *equivalent* if they are *identical*.

?- 3==3.  
true.

?- X==3.  
false.

?- X==Y.  
false.

?- X=3, X==3.  
X = 3.

?- X==Y, X=Y.  
false.

?- X=Y, X==Y.  
X = Y.

different from object/reference equality
Evaluation equality

Two terms are *evaluation equivalent* if they *evaluate* to the same number.

?- 3+1::=4.
true.

?- 3+1::=4.
false.

?- 3+1::=4.
false.

?- X=3, X*2::=X+3.
X = 3.

?- 3::=X.
ERROR: ::=:/2: Arguments are not sufficiently instantiated

?- X*2::=X+3, X=3.
ERROR: ::=:/2: Arguments are not sufficiently instantiated
List construction

Lists are **terms** with special syntax.

\[[3, 4, 5] \equiv 3 \cdot (4 \cdot (5 \cdot (\text{Empty List})))\]

\[[3, [4, 5], 6]\]
### List patterns

- **story([3,little,pigs])**
  - Head: 3
  - Tail: [little,pigs]

- ?- story([X,Y]).
  - X = 3,
  - Y = [little,pigs].

- ?- story([X,Y,Z]).
  - X = 3,
  - Y = little,
  - Z = pigs.

- ?- story([X,Y,Z,V]).
  - X = 3,
  - Y = little,
  - Z = pigs,
  - V = [].

- ?- story([X,Y,Z,V]).
  - false.

---

**Haskell**

- x:y:z
- [x,y,z]

**Prolog**

- [X,Y|Z]
- [X,Y,Z]
List predicates

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

wildcard — matches anything

?- member(3, [2, 3, 4, 3]).
true ;
true.

?- member(3, [2, 3, 4, 3, 1]).
true ;
true ;
false.

?- member(2, [2, 3, 4, 3]).
true ;
false.

?- member(3, L).
L = [3 | A] ;
L = [A, 3 | B] ;
L = [A, B, 3 | C]
...

?- member(X, [3, 4]).
X = 3 ;
X = 4.
Arithmetic in Prolog

Arithmetic expressions are also constructed data (nested predicates)

- special syntax: can be written infix, standard operator precedence
- can be evaluated:

\[
\begin{align*}
X \text{ is } & \text{exp} \quad \text{evaluate exp and bind to } X \\
\text{exp} \text{ =:= } & \text{exp} \quad \text{evaluate expressions and check if equal}
\end{align*}
\]

\[3\times4+5\times6 \equiv +(*(3,4), *(5,6))\]

?- X is 3*4+5*6.
X = 42.

?- 8 is X*2.
ERROR: is/2: Arguments are not sufficiently instantiated

Arithmetic operations
+ - * / mod

Comparison operations
< > =< >= =:= =\=
Using arithmetic in rules

Example database
fac(1,1).
fac(N,M) :- K is N - 1, fac(K,L), M is L*N.

?- fac(5,M).
X = 120.

?- fac(N,6).
ERROR: fac/2: Arguments are not sufficiently instantiated
Unification vs. arithmetic equality

**Unification: A = B**
Find an assignment of variables that makes its arguments syntactically equal

**Arithmetic equality: A =:= B**
Evaluate terms as arithmetic expressions and check if numerically equal

?- X = 3*5.
X = 3*5.

?- 8 = X*2.
false.

?- 4*2 = X*2.
X = 4.

?- X is 3*5.
X = 15.

?- 8 is X*2.
ERROR: is/2: Arguments are not sufficiently instantiated.
Outline

Programming paradigms

Logic programming basics
   Introduction to Prolog
   Predicates, queries and rules

Understanding the query engine
   Goal search and unification
   Structuring recursive rules

Complex terms and lists

Cuts and negation
How cut works

**Cut** is a special atom! used to **prevent backtracking**

When encountered as a subgoal it:
- always succeeds
- commits the current goal search to the matches and assignments made so far

---

**Database without cut**

foo(1). foo(2).
bar(X,Y) :- foo(X), foo(Y).
bar(3,3).

?- bar(A,B).
A = 1, B = 1 ;  A = 1, B = 2 ;
A = 2, B = 1 ;  A = 2, B = 2 ;
A = 3, B = 3 .

**Database with cut**

foo(1). foo(2)
bar(X,Y) :- foo(X), !, foo(Y).
bar(3,3).

?- bar(A,B).
A = 1, B = 1 ;  A = 1, B = 2 .
Green cuts vs. red cuts

A **green cut**: doesn’t affect the members of a predicate
- only cuts paths that would have failed anyway
- the cut is used purely for efficiency

A **red cut**: any cut that isn’t green
- if removed, meaning of the predicate changes
- this cut is part of the “logic” of the predicate

```prolog
max(X, Y, Y) :- X < Y, !.
max(X, Y, X) :- X >= Y.
find(X, [X|_]) :- !.
find(X, [_|L]) :- find(X, L).
```
Negation as failure

**Negation predicate**
not(P) :- P, !, fail.
not(P).

**Example database**
hobbit(frodo).
hobbit(bilbo).
likes(X,beer) :- hobbit(X).

?- not(likes(frodo,beer)).
false.

?- not(likes(gimli,beer)).
true.

?- not(likes(bilbo,X)).
false.

?- not(likes(X,pizza)).
true.