Introduction to Logic Programming in Prolog
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
What is a programming paradigm?

**Paradigm**  
A conceptual model underlying the theories and practice of [a scientific subject](adapted from Oxford American)

**scientific subject = programming**

**Programming paradigm**  
A conceptual model underlying the theories and practice of [programming](programming)
Imperative paradigm

Imperative model

<table>
<thead>
<tr>
<th>data</th>
<th>set of state variables</th>
<th>type State = [(Name, Val)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>computation</td>
<td>transformation of state</td>
<td>State -&gt; State</td>
</tr>
</tbody>
</table>

Needs two sub-languages:

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

Semantic functions:

- semE :: Expr -> State -> Val
- semS :: Stmt -> State -> 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 extend statements with:

- constructs to **create objects** and **invoke methods**

New statement semantic function:

- \( \text{semS} :: \text{Stmt} \rightarrow [\text{Object}] \rightarrow [\text{Object}] \)
### 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>

In general, just one language (e.g. lambda calculus)

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

Semantic function:

- `sem :: Expr -> Val`
### 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>comp.</td>
<td>query over relations</td>
<td>type Query = Known -&gt; Known</td>
</tr>
</tbody>
</table>

In general, just one language:

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

Semantic function:

- **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>
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
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

\[
mortal(X) :- \text{human}(X).
\text{human}(\text{socrates}).
\]

Prolog query (interactive)

\[
?- \text{mortal}(\text{socrates}).
\text{true}.
\]
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
## 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>

All of these except **help** also work in GNU Prolog!
An **atom** is just a primitive value

- string of characters, numbers, underscores starting with a **lowercase letter**:
  - hello, socrates, sUp3r_At0m

- any single quoted string of characters:
  - 'Hello world!', 'Socrates'

- numeric literals: 123, -345

- empty list: []
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

- special variable: _ (just an underscore)
  - unifies with anything – “don’t care”
Basic entity in Prolog is a **predicate**

**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.

```prolog
?- hobbit(sam).
true.

?- hobbit(gimli).
false.

?- 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

likes(bilbo, frodo).
likes(frodo, bilbo).
likes(sam, frodo).
likes(frodo, ring).

?- likes(frodo,Y).
Y = bilbo ;
Y = ring .

?- likes(X,frodo).
X = bilbo ;
X = sam .

?- likes(X,Y).
X = bilbo ,
Y = frodo ;
X = sam ,
Y = frodo ;
Overloading predicates

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

### hobbit/1
- hobbit(bilbo).
- hobbit(frodo).
- hobbit(sam).

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

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

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

**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
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
How does prolog solve queries?

Basic algorithm for solving a (sub)goal

1. Linearly search database 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 subgoals
   - If unification is unsuccessful: keep searching
3. Backtrack if we reach the end of the database
1. Search the database for candidate matches

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 and goal

Unification
Find an **assignment of variables** that makes its arguments **syntactically equal**

Prolog: $A = B$ means attempt to **unify** $A$ and $B$

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

?- likes(merry,Y) = likes(merry,pippin).
Y = pippin .

?- likes(merry,Y) = likes(X,beer).
X = merry ; Y = beer .

2a. if **fail**, try next candidate

2b. if **success**, add new subgoals
Tracking subgoals

Deriving solutions through rules

1. Maintain a list of goals that need to be solved
   - when this list is empty, we’re done!
2. If current goal unifies with a rule **head**, add **body** as subgoals to the list
3. After each unification, **substitute variables** in all goals in the list!

Database

1. \(lt(\text{one, two})\).
2. \(lt(\text{two, three})\).
3. \(lt(\text{three, four})\).
4. \(lt(X,Z) :- lt(X,Y), lt(Y,Z)\).  

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

1. \(Y_1=\text{two}\)
2. \(Y_2=\text{three}\)
3. \(\text{true}\)
4: \(X=\text{one}, Z=\text{four}\)  
   \(lt(\text{one, four})\)
4: \(X=\text{two}, Z=\text{four}\)  
   \(lt(Y_1, four)\)
4: \(X=\text{two}, Z=\text{four}\)  
   \(lt(\text{one, Y_1}), lt(Y_1, four)\)
4: \(X=\text{two}, Z=\text{four}\)  
   \(lt(\text{two, four})\)
4: \(X=\text{two}, Z=\text{four}\)  
   \(lt(\text{two, Y_2}), lt(Y_2, four)\)
4: \(X=\text{two}, Z=\text{four}\)  
   \(lt(\text{three, four})\)
4: \(X=\text{two}, Z=\text{four}\)  
   \(done!\)
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
Potential for infinite search

Why care about how goal search works?

One reason: to write **recursive rules** that don’t loop!

**Bad example: symmetry**

married(abe, mona).
marrried(clancy, jackie).
marrried(homer, marge).
marrried(X, Y) :- married(Y, X).

?- married(jackie, abe).
ERROR: Out of local stack

**Bad example: transitivity**

lt(one, two).
l(two, three).
l(two, 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** (in database and rule bodies)
2. Use helper predicates to **enforce progress** during search

Example: symmetry

marriedP(abe, mona).
marrriedP(clancy, jackie).
marrriedP(homer, marge).
marrried(X, Y) :- marriedP(X, Y).
marrried(X, Y) :- marriedP(Y, X).

?- married(jackie, abe).
false.

Example 2: transitivity

ltP(one, two).
ltp(two, three).
ltp(three, four).
ltp(X, Y) :- ltP(X, Y).
ltp(X, Z) :- ltP(X, Y), lt(Y, Z).

?- 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
Representing structured data

Can represent structured data by **nested predicates**

**Example database**

- drives(bart, skateboard(green)).
- drives(bart, bike(blue)).
- drives(lisa, bike(pink)).
- drives(homer, car(pink)).

?- drives(lisa, X).
  X = bike(pink).

?- drives(X, bike(Y)).
  X = bart, Y = blue;  
  X = lisa, Y = pink.

Variables can’t be used for predicates:

?- drives(X, Y(pink)). ← illegal!