Programming in Logic: Prolog

Prolog’s Declarative & Procedural Semantics

Readings: Sections 2.3 - 2.7
Review

- Data objects either constants or variables

- Variables:
  - Either bound or unbound
  - Bound either explicitly via matching or implicitly via goal matching
  - Once bound, value cannot be unbound or changed except through backtracking.
Review cont’d

- Structures have functor name and arity.
- Matching is straightforward except when matching unbound variable against structure containing occurrence of that variable.
  - Then effect of matching depends upon whether that version of Prolog implemented the occurs check
    - If it did then match fails
    - If not then match succeeds creating an “infinite” data structure.
Where is the Prolog Program?

- When one talks about Prolog programs, one is referring to the Knowledge base.
- The definitions of the relations in the Knowledge base is the Prolog code.
Goals

- Goals are the terms that Prolog attempts to match against the clause heads in its Knowledge base.

- The top level terms in a query are all goals:
  - `?: a(4, x), b(Y, 7).

- The top level terms in the body of a clause are all goals:
  - `p(X) :- q(X, a), r(3, X).`
Goals and Data Structures

- Prolog goals look exactly like Prolog data structures.
- In other words, Prolog code looks just like Prolog data. *This is intentional!*
- A Prolog program can modify itself as it is running.
- A Prolog program can create new queries & pose them to the Prolog interpreter.
Prolog’s Declarative Semantics

- The prolog we have seen so far is called *pure* Prolog, i.e., everything can be expressed in FOPC.

- For example:

  \[\text{ancestor}(A,P) :- \text{parent}(A,X), \text{ancestor}(X,P).\]

  can be translated into the logical expression:

  \[\forall A \forall P (\text{ancestor}(A,P) \iff \text{parent}(A,X) \land \text{ancestor}(X,P))\]
Translating Prolog clauses into FOPC

The clauses:

\[ \text{ancestor}(A,P) \leftarrow \text{parent}(A,P). \]

\[ \text{ancestor} (A,P)\leftarrow \text{parent}(A,X), \text{ancestor}(X,P). \]

Translates into FOPC as:

\[ \forall A \forall P ( \text{ancestor}(A,P) \leftarrow \text{parent}(A,P) \lor (\text{parent}(A,X) \land \text{ancestor}(X,P))) \]
Prolog Declarative Semantics

- In the body of the clause, the commas separating the top-level terms are understood to be conjunctions (and’s).
- There are two ways to specify disjunctions (or’s):
  - Write separate clauses (as done for ancestor)
  - Use a semicolon (“;”)

Disjunctions in Prolog

We could rewrite the ancestor example in Prolog as follows:

\[
\text{ancestor}(A,P) \leftarrow \text{parent}(A,P); \\
\text{parent}(A,X), \text{ancestor}(X,P).
\]
Negation in Pure Prolog

- We cannot express negation in the clause body in pure prolog:
  - For example, we can’t express in pure Prolog the following FOPC expression
    \[ \forall H \forall W ( \text{husband}(H, W) \leftarrow \text{married}(H, W) \land \text{male}(H) \land \neg \text{divorced}(H, W) ) \]
Negation in Pure Prolog cont’d

- Likewise, Prolog can’t tell us something is false rather it tells us that it can’t prove something is true.

- For example, given the Knowledge base:
  - `man(socrates).`
  - `mortal(X) :- man(X).`

- The answer to `mortal(mike)` would be **no**, not because it isn’t true, but because Prolog can’t prove it’s true using that KB.
Prolog’s Procedural Semantics I

- Prolog’s control flow different from languages like Java.

- In Prolog no distinction between statements & method calls - all goals recursively matched against KB (system defined relations (write/1) have own definitions).

- Goal parameters can be unbound variables.

- Goals succeed or they fail.
Prolog Control Flow Example

- Given the query \( a(2) \), Prolog scans the KB for the first clause head that matches the first query goal.

- If it finds a match then it recurses on the goals in the clause body until it bottoms out.

- Query succeeded here.

**KB**

1. \( a(X) :- b(X) \).

2. \( b(2) \).

**Execution trace of query**

- \([a(2)]\) (goal stack)

- 1. \( X = 2 \) (which clause \([b(2)]\) matched & bindings)

- 2. \([\ ]\)
Prolog Control Flow Example

**Query:** \( a(2) \)

Notice that after the query goal matched the clause head, the query goal \((a(2))\) is replaced by the first clause body goal \((b(2))\) in the next step of the trace with its variable bound.

**KB**

1. \( a(X) :- b(X). \)
2. \( b(2). \)

**Execution trace of query**

\([a(2)]\) (goal stack)

1. \( X = 2 \) (which clause \([b(2)]\) matched & bindings)
2. [ ]
Prolog Control Flow Example

**Query**: $a(3)$, Prolog scans the KB for first clause head matching first query goal.

- If it fails to find a match then it tries to backtrack its matches.
- There are no other possible matches.
- Query fails.

**KB**

1. $a(X) :- b(X)$.
2. $b(2)$.

**Execution trace of query**

$$\begin{align*}
[a(3)] \\
1. X = 3 \\
[b(3)] \\
\times
\end{align*}$$
Backtracking

- **Query:** $a(3)$, from point of failure, Prolog backtracks to last match (matches clause 1), Prolog continues its scan of KB for another clause head that matches goal.

- Matches clause 2 and succeeds.

**KB:**
1. $a(X) :- b(X)$.
2. $a(3)$.
3. $b(2)$.

**Execution trace of query**

```
[a(3)]
1. X = 3
2. [b(3)]
```

$\times$
Flow of Control with Conjunctions

Query: $a(X)$.  

KB: 
1. $a(X) :- b(X), c(X)$.  
2. $b(2)$.  
3. $c(2)$.  

Response: 
$X = 2$ ?

$[a(X)]$  

Note renaming of variables in matched clause, $X$ replaced by $X'$.  

$[c(2)]$  

Goal query is replaced by first clause body goal and rest of clause body goals are pushed onto goal stack.
Flow of Control with Conjunctions

Query: 
\[ a(X). \]

KB: 
1. \[ a(X) :- b(X), c(X). \]
2. \[ b(2). \]
3. \[ b(3). \]
4. \[ c(3). \]

Response: 
\[ X = 3 ? \]

Notice unbinding of variable (X’) when backtrack over binding point.
Prolog Procedural Semantics II

- When goal fails, control undoes any bindings done at that step & backs up to previous step.
- It continues scan of KB for clause heads matching current goal.
- If none found then continues to backtrack.
Flow of Control with Conjunctions

**Query:**
a(X).

**KB:**
1. a(X) :- b(X), c(X).
2. b(Y) :- d(Z), e(Y,Z).
3. b(3)
4. c(3).
5. d(5).
6. e(3,5).

**Response:**
X = 3 ?

\[
\begin{align*}
[a(X)] \\
1. &X = X' \\
[b(X'), c(X')] \\
2. &X' = Y \\
[d(Z), e(Y,Z), c(Y)] \\
5. &Z = 5 \\
[e(Y,5), c(Y)] \\
6. &Y = 3 \\
[c(3)] \\
4. & \\
[]
\end{align*}
\]
Flow of Control with Conjunctions

**Query:**
\[a(X)\]

**KB:**
1. \[a(X) :- b(X), c(X)\]
2. \[b(Y) :- d(Z), e(Y,Z)\]
3. \[b(3)\]
4. \[c(4)\]
5. \[d(5)\]
6. \[e(3,5)\]
7. \[e(4,5)\]

**Response:**
\[X = 4 ?\]
Additional Answers

- What happens when Prolog returns an answer and prompts you with “?”?
- Since an answer was returned, the trace ended with a true and an empty goal stack.
- If you type in “;” then Prolog will, in effect, change that true to fails and backtrack.
A Prolog program can be declaratively correct, but procedurally inadequate (may recurse forever ($p :- p.$), or simply be inefficient).

The program’s behavior is determined by the order of the clauses in the KB and the order of the goals in the clauses.

Changing either can dramatically affect the behavior of the program.
Example: Parental Fact KB

- parent(pam,bob).
- parent(tom,bob).
- parent(tom,liz).
- parent(bob,ann).
- parent(bob,pat).
- parent(pat,jim).
Predecessor Relation Variations

\[\text{pred1}(X,Z) :\text{-} \text{parent}(X,Z).\]
\[\text{pred1}(X,Z) :\text{-} \text{parent}(X,Y), \text{pred1}(Y,Z).\]

\[\text{pred2}(X,Z) :\text{-} \text{parent}(X,Y), \text{pred2}(Y,Z).\]
\[\text{pred2}(X,Z) :\text{-} \text{parent}(X,Z).\]

\[\text{pred3}(X,Z) :\text{-} \text{parent}(X,Z).\]
\[\text{pred3}(X,Z) :\text{-} \text{pred3}(Y,Z), \text{parent}(X,Y).\]

\[\text{pred4}(X,Z) :\text{-} \text{pred4}(Y,Z), \text{parent}(X,Y).\]
\[\text{pred4}(X,Z) :\text{-} \text{parent}(X,Z).\]
Computing \( \text{predn}(\text{tom}, \text{pat}) \)

- For \( n \) from 1 to 3, Prolog returns \textit{yes}.
- For \( \text{pred}4(\text{tom}, \text{pat}) \), Prolog loops forever.
- While the first 3 \textit{preds} find an answer they do so with different amounts of computation.
Trace of $predl(tom, pat)$

KB:

1. $parent(pam, bob)$.  
2. $parent(tom, bob)$.  
3. $parent(tom, liz)$.  
4. $parent(bob, ann)$.  
5. $parent(bob, pat)$.  
6. $parent(pat, jim)$.  
7. $predl(X, Z) :- parent(X, Z)$.  
8. $predl(X, Z) :- parent(X, Y), predl(Y, Z)$.

$[predl(tom, pat)]$

7. $X=tom, Z=pat$  
8. $X=tom, Z=pat$

$[parent(tom, pat)]$  
$[parent(tom, Y), predl(Y, pat)]$

$x$  
$2. Y=bob$

$[predl(bob, pat)]$  

$7. X'=bob, Z'=pat$

$[parent(bob, pat)]$

$[ ]$

$[ ]$
Exercises

Try doing the execution traces of
\textit{pred2(tom,pat)}, \textit{pred3(tom,pat)}, and
\textit{pred4(tom,pat)}
Quick Quiz

- What is the difference between the structure of Prolog code and Prolog data?
- What is a goal and how is it different from structured data?
- How do you express a negated goal?
- What does it mean when Prolog answers no
Quick Quiz cont’d

- In pure Prolog, does changing the order of the clauses or of the goals in the clauses change the declarative meaning?
- What about the procedural meaning?
- Is there anything declaratively wrong with:
  - \texttt{ancestor(A,P) :- ancestor(X,P), parent(A,X).}
  - \texttt{ancestor(A,P) :- parent(A,P).}
- Is there anything procedurally wrong?
Summary

- Declarative semantics of pure Prolog program defines when a goal is true and for what instantiation of variables.
- Commas between goals mean \textit{and}, semicolons mean \textit{or}.
- Procedural semantics are determined by clause and goal ordering.
- Goal failure causes backtracking and unbinding of affected variables.