Introduction to Prolog

Notes for CSCE 330

Based on Bratko and Van Emden

Marco Valtorta

A Little History

- Prolog was invented by Alain Colmerauer, a professor of computer science at the university of Aix-Marseille in France, in 1972
- The first application of Prolog was in natural language processing
- Prolog stands for programming in logic (PROgrammation en LOgique)
- Its theoretical underpinning are due to Donald Loveland of Duke university through Robert Kowalski (formerly) of the university of Edinburgh

Logic Programming

- Prolog is the only successful example of the family of logic programming languages
- A Prolog program is a theory written in a subset of first-order logic, called Horn clause logic
- Prolog is declarative. A Prolog programmer concentrates on *what* the program needs to do, not on *how* to do it
- The other major language for Artificial Intelligence programming is LISP, which is a functional (or applicative) language

Prolog in CSCE 580 (Skip for 330)

- Chapters 1-9 (the Prolog programming language), 11 (blind search), 12 (heuristic search), and *maybe* 14 (constraint logic programming) and some parts of 15 (Bayesian networks), 17 (means-ends analysis) and 18 (induction of decision trees)
- Prolog is introduced as a programming language *before* a thorough review of first-order logic

Defining Relations by Facts

- parent(tom,bob).
- parent is the name of a relation
  - A relation of arity n is a function from n-tuples (elements of a Cartesian product) to {true, false}. (It can also be considered a subset of the n-tuples.)
- parent(pam, bob). parent(tom,bob). parent(tom,liz). parent(bob, ann). parent(bob,pat). parent(pat,jim).
- A relation is a collection of *facts*

Queries

?-parent(bob,pat).

yes

- A query and its answer, which is correct for the relation defined in the previous slide: this query *succeeds*

?-parent(liz,pat).

no

- A query and its answer, which is correct for the relation defined in the previous slide: this query *fails*

More Queries

- cf. pr1_1.pl

```
?-parent( tom,ben).      /* who is Ben? */
?-parent( X,liz).        /* Wow! */
?-parent( bob,X). /* Bob's children */
?-parent( X,Y).  /* The relation, fact by fact */
```

Composite Queries

- Grandparents:

  ?-parent(Y,jim), parent(X,Y).

  - the comma stands for “and”

  ?-parent(X,Y), parent(Y,jim).

  - order should not matter, and ***it does not!***
- Grandchildren:

  ?-parent(tom,X), parent(X,Y).
- Common parent, i.e. (half-)sibling:

  ?-parent(X,ann), parent(X,pat).

Facts and Queries

- Relations and queries about them
- Facts are a kind of *clause*
  - *Prolog programs consist of a list of clauses*
- The arguments of relations are atoms or variables (a kind of *term*)
- Queries consist of one or more *goals*
- Satisfiable goals succeed; unsatisfiable goals fail

Defining Relations by Rules

- The offspring relation:

  For all X and Y,

  Y is an offspring of X if

  X is a parent of Y

- This relation is defined by a rule, corresponding to the Prolog clause

  offspring(Y,X) :- parent(X,Y).

- Alternative reading:

  For all X and Y,

  if X is a parent of Y,

  then Y is an offspring of X

Rules

- Rules are clauses. Facts are clauses
- A rule has a condition and a conclusion
- The conclusion of a Prolog rule is its *head*
- The condition of a Prolog rule is its *body*
- If the condition of a rule is true, then it follows that its conclusion is true also

How Prolog Rules are Used

- Prolog rules may be used to define relations
- The offspring relation is defined by the rule offspring(Y,X) :- parent(X,Y):
  - if (X,Y) is in the parent relation, then (Y,X) is in the offspring relation
- When a goal of the form offspring(Y,X) is set up, the goal succeeds if parent(X,Y) succeeds
- Procedurally, when a goal matches the head of a rule, Prolog sets up its body as a new goal

Example (ch1_2.pl)

?-offspring(liz,tom).

- No fact matches this query
- The head of the clause

  offspring(Y,X) :- parent(X,Y) does
- Y is replaced with liz, X is replaced with tom
- The instantiated body *parent(tom,liz)* is set up as a new goal
- ?-parent(tom,liz) succeeds
- offspring(liz,tom) therefore succeeds too

More Family Relations

- *female* and *male* are defined *extensionally*, i.e., by facts; *mother* and *grandparent* are defined *intensionally*, I.e., by rules
- female(pam). … male(jim).
- mother(X,Y) :- parent(X,Y), female(X).
- grandparent(X,Z) :- parent(X,Y), parent(Y,Z).

Sister (ch1_3.pl)

- sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X).
- Try:

  ?-sister(X,pat).

  X = ann;

  X = pat /* Surprise! */
- (Half-)sisters have a common parent *and* are different people, so the correct rule is:
- sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X), different(X,Y).
  - (or: sister(X,Y) :- parent(Z,X), parent(Z,Y), parent(W,X), parent(W,Y), female(X), different(Z,W), different(X,Y).)

Clauses and Instantiation

- Facts are clauses without body
- Rules are clauses with both heads and non-empty bodies
- Queries are clauses that only have a body (!)
- When variables are substituted by constants, we say that they are *instantiated*.

Universal Quantification

- Variables are universally quantified, but beware of variables that only appear in the body, as in *haschild(X) :- parent(X,Y).*, which is best read as:

for all X,

X has a child if

there exists some Y such that X is a parent of Y

- (I.e.: for all X and Y, if X is a parent of Y, then X has a child)

Predecessor (Ancestor)

- predecessor(X,Z) :- parent(X,Z).
- predecessor(X,Z) :- parent(X,Y), parent(Y,Z).
- predecessor(X,Z) :- parent(X,Y1),
  parent(Y1,Y2,),
  parent(Y2,Z).

etc.

- When do we stop?
- The length of chain of people between the predecessor and the successor should not arbitrarily bounded.

Note on History in SWI-Prolog

- See p.24 of manual (query substitution)
- To set up the history mechanism, edit the pr.ini file and place it in one of the directories in file_search_path. (I placed it in the directory where my Prolog code is.)
- To check the values of Prolog flags, use

  ?-current_prolog_flag(X,Y).

Note on Definition Graphs
Skip for 330

- Definition graphs indicate that definition of relations by rules is somewhat analogous to function composition in applicative (functional) languages
- See Figures 1.3 and 1.4.
- "each diagram should be understood as follows: if the relations shown by solid arcs hold, the relation shown by a dashed arc also holds"

A Recursive Rule

- For all X and Z,

  X is a predecessor of Z if

  there is a Y such that

  (1) X is a parent of Y and

  (2) Y is a predecessor of Z.

- predecessor(X,Z) :-

  parent(X,Y),

  predecessor(Y,Z).

The Family Program (fig1_8.pl)

- Comments
  - /* This is a comment */
  - % This comment goes to the end of the line
- SWI Prolog warns us when the clauses defining a relation are not contiguous.

Prolog Proves Theorems

- Prolog accepts facts and rules as a set of axioms, and the user's query as a conjectured theorem. Prolog then tries to prove the theorem, i.e., to show that it can be logically derived from the axioms
- Prolog builds the proof "backwards": it does not start with facts and apply rules to derive other facts, but it starts with the goals in the user's query and replaces them with new goals, until new goals happen to be facts

Goal Trees

- In attempting to prove theorems starting from goals, Prolog builds *goal trees*
- See example of proving predecessor(tom,pat) in Figures 1.9, 1.10, 1.11.
- Variables are matched as new goals are set up
- The scope of each variable is a single clause, so we rename variables for each rule application
- Prolog backtracks as needed when a branch of the proof tree is a dead end
- This is explained in more detail in Chapter 2

Declarative and Procedural Meaning of Prolog Programs

- The declarative meaning is concerned with the relations defined by the program: what the program states and logically entails
- The procedural meaning is concerned with how the output of the program is obtained, i.e., how the relations are actually evaluated by the Prolog system
- It is best to concentrate on the declarative meaning when writing Prolog programs
- Unfortunately, sometimes the programmer must also consider procedural aspect (for reasons of efficiency or even correctness): we will see examples of this in Ch.2

Knight Moves on a Chessboard

- % This example is from unpublished (to the best of my knowledge) notes by Maarten
- % Van Emden.
- /* The extensional representation of the (knight) move relation follows. It
- consists of 336 facts; only a few are shown. In particular, all moves from
- position (5,3) on the chess board are shown. */

- move(1,1,2,3).
- move(1,1,3,2).
- ....
- move(5,3,6,5).
- move(5,3,7,4).
- move(5,3,7,2).
- move(5,3,6,1).
- move(5,3,4,1).
- move(5,3,3,2).
- move(5,3,3,4).
- move(5,3,4,5).
- ...
- move(8,8,7,6).
- move(8,8,6,7).

Intensional Representation of Moves

- /* The intensional representation of the (knight) move relation follows. It
- consists of facts (to define extensionally the relation succ/2) and rules (to
- define the relations move, diff1, and diff2. */

- move(X1,Y1,X2,Y2) :- diff1(X1,X2), diff2(Y1,Y2).
- move(X1,Y1,X2,Y2) :- diff2(X1,X2), diff1(Y1,Y2).

- diff1(X,Y) :- succ(X,Y).
- diff1(X,Y) :- succ(Y,X).

- diff2(X,Z) :- succ(X,Y), succ(Y,Z).
- diff2(X,Z) :- succ(Z,Y), succ(Y,X).

- succ(1,2).
- succ(2,3).
- succ(3,4).
- succ(4,5).
- succ(5,6).
- succ(6,7).
- succ(7,8).