An Example Class on the Use of Prolog as a logic programming tool

Learning Objective: To introduce Prolog as a tool to programming in Logic, - Propositional and Predicate Logic.

Learning Outcome:

a) Understand the separation of Knowledge from the inference mechanism in Knowledge-Based System (KBS).

b) Understand that the inference process can be represented in the form of AND-OR tree; where AND branches are the premises and OR branches are the alternative matching in the KBS.

c) Appreciate the structure of the AND-OR is heavily influenced by the ordering of the facts and rules in the KBS.

d) Understand the process of unification in zero and first order logic.

Tools: SWI-Prolog - http://www.swi-prolog.org/

Introduction:

1. How to Run Prolog

SWI-Prolog's website has lots of information about SWI-Prolog, a download area, and documentation. The upkeep for SWI-Prolog is excellent. The link ...

SWI-Prolog Home Page (current as of March 2013)

The examples in this tutorial use a simplified form of interaction with a *typical* Prolog interpreter. The sample programs should execute similarly on any system using an Edinburgh-style Prolog interpreter or interactive compiler.

Under Windows, SWI-Prolog installs a start icon that can be double-clicked to initiate the interpreter. The interpreter then starts in its own command window.

A startup message or banner may appear, and that will soon be followed by a goal prompt looking similar to the following

```
?- _
```

Interactive *goals* in Prolog are entered by the user following the '?- ' prompt.

Many Prologs have command-line help information. SWI Prolog has extensive help information. This help is indexed and guides the user. To learn more about it, try

```
?- help(help).
```

Notice that all of the displayed symbols need to be typed in, followed by a carriage return.

To illustrate some particular interactions with prolog, consider the following sample session. Each file referred to is assumed to be a local file in the user's account, which was either created by the user, obtained by copying directly from some other public source, or obtained by saving a text file while using a web browser. The way to achieve the latter is either to follow a URL to the source file and then save, or to select text in a Prolog Tutorial web page, copy it, paste into a text editor window and then save to file. The comments /* ... */ next to goals are referred to in the notes following the session.

```
?- ['2_2.pl'].          /* 1. Load a program from a local file*/
yes
?- listing(factorial/2). /* 2. List program to the screen*/

factorial(0,1).

factorial(A,B) :-
        A > 0,
        C is A-1,
        factorial(C,D),
        B is A*D.
yes

?- factorial(10,What).     /* 3. Compute factorial of 10 */
What=3628800

?- ['2_7.pl'].          /* 4. Load another program */

?- listing(takeout).

takeout(A,[A|B],B).
takeout(A,[B|C],[B|D]) :-
       takeout(A,C,D).
yes

?- takeout(X,[1,2,3,4],Y).  /* 5. Take X out of list [1,2,3,4] */
X=1  Y=[2,3,4] ;             Prolog waits ... User types ';' and Enter
X=2  Y=[1,3,4] ;              again ...
X=3  Y=[1,2,4] ;              again ...
X=4  Y=[1,2,3] ;              again ...
no                        Means: No more answers.

?- takeout(X,[1,2,3,4],_), X>3.  /* 6.  Conjunction of goals */
X=4 ;
no

?- halt.                    /* 7. Return to OS */
```

The comments appearing at the right at various spots in a sample session were added with a text processor. They also serve as reference signposts for the notes which appear below. We discuss several points now, while other details will be deferred to later sections.

Notes:

1. A Prolog goal is terminated with a period "." In this case the goal was to load a program file. This "bracket" style notation dates back to the first Prolog implementations. Several files can be chain loaded by listing the filenames sequentially within the brackets, separated by commas. In this case, the file's name is 2_1.pl (programs corresponding to Section 7.1 of this tutorial), which contains two prolog programs for calculating the factorial of a positive integer. The actual program in the file is discussed in Section 2.1. The program file was located in the current directory. If it had not been, then the path to it would have to have been specified in the usual way.

2. The built-in predicate 'listing' will list the program in memory -- in this case, the factorial program. The appearance of this listing is a little different than the appearance of the source code in the file, which we will see in Section 2.1. Actually, Quintus Prolog compiles programs unless predicates are declared to be dynamic. Compiled predicates do not have an interactive source listing that can be supplied by a 'listing' goal. So, in order to illustrate this Prolog interpreter feature, the predicates were declared as dynamic in the source code before this sample run.

3. The goal here, 'factorial(10,What)', essentially says "the factorial of 10 is What?". The word 'What' begins with an upper-case letter, denoting a logical variable. Prolog satisfies the goal by finding the value of the variable 'What'.

4. Both "programs" now reside in memory, from the two source files 2_2.pl and 2_7.pl. The 2_7.pl file has many list processing definitions in it. (See Section 2.7.)

5. In the program just loaded is a definition of the logical predicate 'takeout'. The goal 'takeout(X,[1,2,3,4],Y)' asks that X be taken out of list [1,2,3,4] leaving remainder list Y, in all possible ways. There are four ways to do this, as shown in the response. The 'takeout' predicate is discussed in Section 2.7. Note, however, how Prolog is prodded to produce all of the possible answers: After producing each answer, Prolog waits with a cursor at the end of the answer. If the user types a semicolon ';' , Prolog will look for a next answer, and so on. If the user just hits Enter, then Prolog stops looking for answers.

6. A compound or conjunctive goal asks that two individual goals be satisfied. Note the arithmetic goal (built-in relation), 'X>3'. Prolog will attempt to satisfy these goals in the left-to-right order, just as they would be read. In this case, there is only one answer. Note the use of an anonymous variable '_' in the goal, for which no binding is reported ("don't-care variable").

7. The 'halt' goal always succeeds and returns the user to the operating system.

Prolog as a Knowledge Base System

The prolog knowledge contains 2 types of knowledge sentences; namely: facts and rules. The inference mechanism is separated from the KB and is part of the Prolog engine. Facts can be propositional symbols or predicate terms.

Facts can be as simple as:

'It is raining today'.

or

jill.

Useful facts usually contain predicates:

```
boy(jack).
girl(jill).
friends(jack, jill).
go(jack, jill, 'up the hill').
give(jack, jill, crown).
```

Names of constants and predicates begin with a lower case letter. The predicate (attribute or relationship, if you will) is written first, and the following objects are enclosed by a pair or parenthesis and separated by commas. Every fact ends with the period character ".".
Order is generally speaking arbitrary, but once you decide on the order, you should be consistent. For example:

```
eating(vladimir, burger).
```

intuitively means that "Vladimir is eating a burger". We could have chosen to put the object of eating (i.e. food) first:

```
eating(burger, vladimir).
```

which we can interpret as "A burger is being eaten by Vladimir". The order is arbitrary in that sense. Rule of thumb is to use 'intuitive' order, sticking to the English language when possible.

These are used in the formulation of rules; which has the following form: if "premise 1" and "premise 2" then "conclusion" and is written in prolog in the following format:

```
characteristicA(anthony).  – an unary property governing the term anthony.
characteristicB(X).        – an unary property governing the variable X.
anthony.                   - propositional term.
```

"conclusion" :- "premise 1", "premise 2".

Facts: male(charles)
female(diana)

Rules:
```
possible_couple(X, Y) :- male(X), female(Y).
possible_couple(X, Y) :- female(X), male(Y).
```

These two rules are alternative ways to define the possibility of being a couple.

Rules are used to express dependency between a fact and another fact:

```
child(X, Y) :- parent(Y, X).
odd(X) :- not even(X).
```

or a group of facts:

```
son(X, Y) :- parent(Y, X) , male(X).
child(X, Y) :- son(X, Y) ; daughter(X, Y).
```

2.4 Loading programs, editing programs

The standard Prolog predicates for loading programs are 'consult', 'reconsult', and the bracket loader notation '[...]'. For example, the goal

```
?- consult('lists.pro').
```

opens the file lists.pro and loads the clauses in that file into memory.

There are two main ways in which a prolog program can be deficient: either the source code has syntax errors, in which case there will be error messages upon loading, or else there is a logical error of some sort in the program, which the programmer discovers by testing the program. The current version of a prolog program is usually considered to be a prototype for the correct version in the future, and it is a common practice to edit the current version and reload and retest it. This *rapid prototyping appraoch* works nicely provided that the programmer has devoted sufficient time and effort to analyzing the problem at hand. Interestingly, if the rapid prototyping approach seems to be failing, this is an excellent signal to take up paper and pencil, rethink the requirements, and start over!

We could call our editor from within prolog ...

```
?- edit('lists.pro'). %% User defined edit, see below ...
```
and then upon returning from the editor (and assuming that the new version of the file was resaved using the same file name), one could use the goal
```
? reconsult('lists.pro').
```

to reload the program clauses into memory, automatically replacing the old definitions. If one had used 'consult' rather than 'reconsult' the old (and possibly incorrect) clauses would have remained in memory along with the new clauses (... this depends upon the Prolog system, actually).

If several files have been loaded into memory, and one needs to be reloaded, use 'reconsult'. If the reloaded file defines predicates which are not defined in the remaining files then the reload will not disturb the clauses that were originally loaded from the other files.

The bracket notation is very handy. For example,

```
?- ['file1.pro',file2.pro',file3.pro'].
```

would load (effectively reconsult) all three files into prolog memory.

Many prolog systems rely upon the programmer to hava a favorite text editor for editing prolog programs. Here is a sample prolog program which calls TextEdit on the Mac(OSX). (This is just an illustration; we do not use TextEdit routinely for writing prolog programs.)

```
edit(File) :-
  name(File,FileString),
  name('open -e ', TextEditString), %% Edit this line for your favorite editor
  append(TextEditString,FileString,EDIT),
  name(E,EDIT),
  shell(E).
```

To use this editor, its source must be loaded (assume it's local to prolog session) ...

```
?- [edit].

yes
```

and then an 'edit' goal can be used (again assume that the file to edit is local to the prolog session)...

```
?- edit('p.pl').
```

{ TextEdit starts up with file loaded. Edit the program... Save the program using the same filename ...}

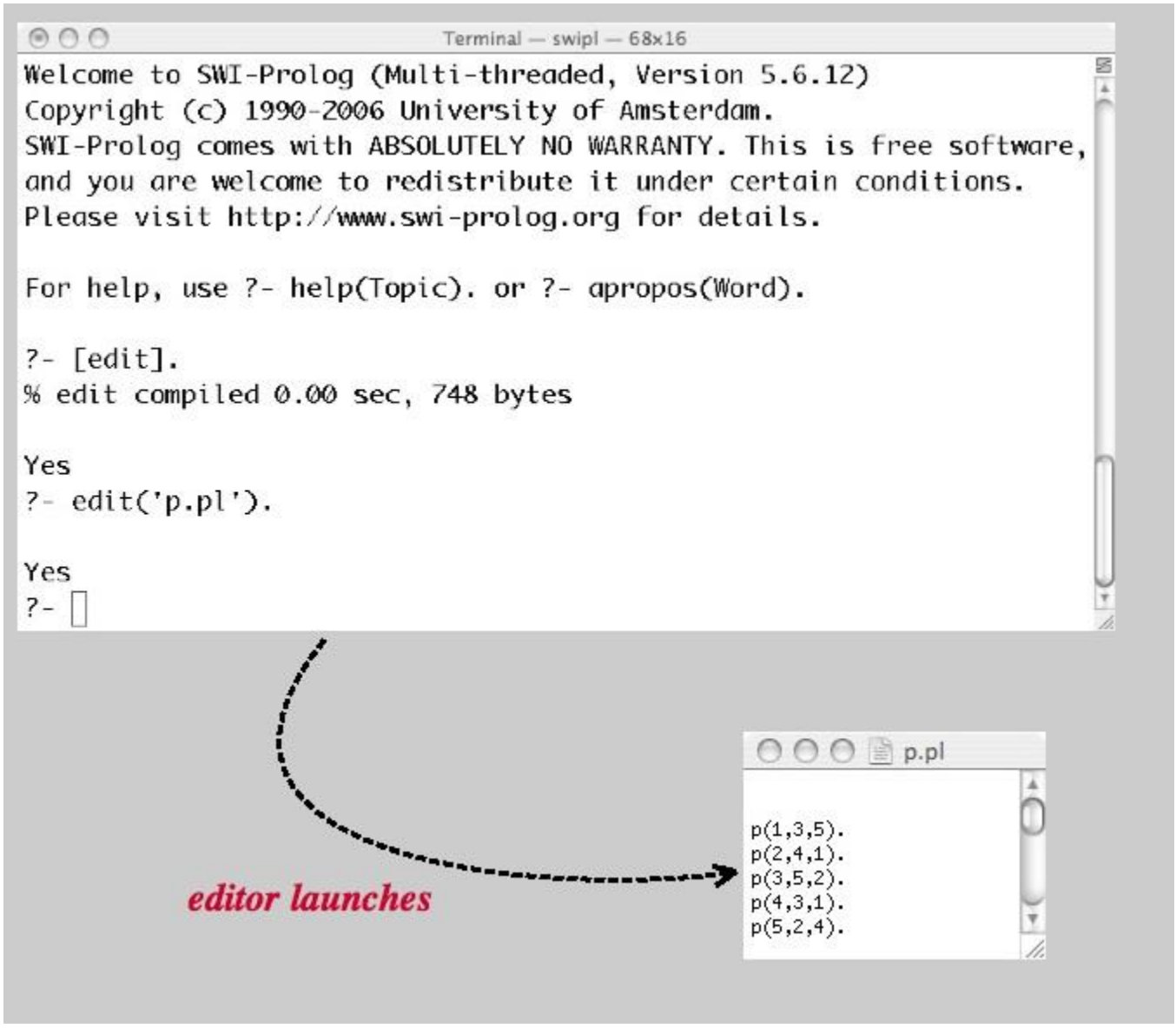

Fig. 2.4.1 Calling external editor

After editing and saving the prolog program, we can reconsult the new version in the prolog session ...

?- reconsult('p.pl').

{ Our prolog session reloads the program for further testing ...}

It is possible to modify the little edit program to suit the user's particular circumstances (various prologs, various operating systems, various editors).

To load clauses supplied interactively by the user, use the goals

```
?-consult(user).
?-reconsult(user).
?-[user].
```

The user then types in clauses interactively, using stop '.' at the end of clauses, and ^Z to end input.

Exercise 2.4 Analyze how the edit program works. First, try goals ...

```
?-name('name',NameString).
```
and
```
?- name(Name,"name")
```

Prolog derivation trees, choices, and unification

To illustrate how Prolog produces answers for programs and goals, consider the following simple datalog program (no functions).

```
/* program P                    clause #      */

p(a).                        /* #1 */
p(X) :- q(X), r(X).             /* #2 */
p(X) :- u(X).                /* #3 */

q(X) :- s(X).                /* #4 */


r(a).                        /* #5 */
r(b).                        /* #6 */


s(a).                        /* #7 */
s(b).                        /* #8 */
s(c).                        /* #9 */

u(d).                         /* #10 */
```

i) Load program P into Prolog and observe what happens for the goal ?-p(X). When an answer is reported, hit (or enter) ';' so that Prolog will continue to trace and find all of the answers.

ii) Load program P into Prolog, turn on the trace, and record what happens for the goal ?-p(X). When an answer is reported, hit (or enter) ';' so that Prolog will continue to trace and find all of the answers. (Use `?- help(trace).` first, if needed.)

The following diagram shows a complete Prolog *derivation tree* for the goal ?-p(X). The edges in the derivation tree are labeled with the clause number in the source file for program P that was used to replace a goal by a subgoal. The direct descendants under any

goal in the derivation tree correspond to *choices*. For example, the top goal `p(X)` unifies with the heads of clauses #1, #2, #3, three choices.

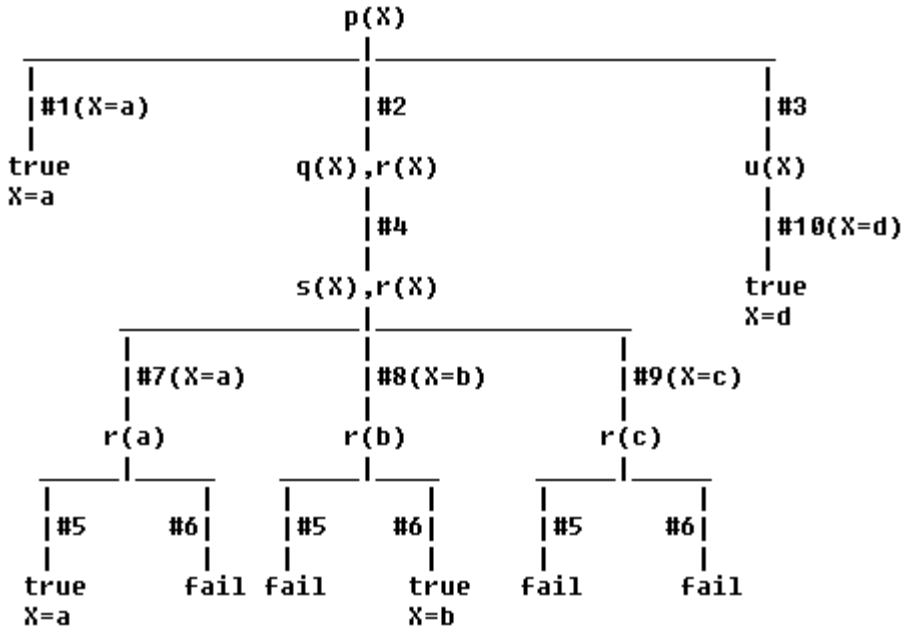

Fig. 3.1.1

The trace (ii above) of the goal ?-p(X) corresponds to a depth-first traversal of this derivation tree. Each node in the Prolog derivation tree was, at the appropriate point in the search, the current goal. Each node in the derivation tree is a sequence of subgoals. The edges directly below a node in this derivation tree correspond to the choices available for replacing a selected subgoal. The current *side clause*, whose number labels the arc in the derivation tree, is tried in the following way: If the leftmost current subgoal (shown as g1 in the little diagram below) unifies with the head of the side clause (shown as h in the diagram), then that leftmost current subgoal is replaced by the body of the side clause (shown as b1,b2,...,bn in the diagram). Pictorially, we could show this as follows:

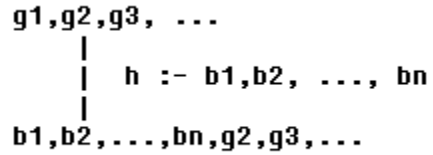

Fig. 3.1.2

One important thing not shown explicitly in the diagram is that the logical variables in the resulting goal b1,b2,...,bn,g2,g3,... have been bound as a result of the unification, and Prolog needs to keep track of these unifying substitutions as the derivation tree grows down any branch.

Now, a depth first traversal of such a derivation tree means that alternate choices will be attempted as soon as the search returns back up the tree to the point where the alternate choice is available. This process is called *backtracking*.

Of course, if the tail of the rule is empty, then the leftmost subgoal is effectively erased. If all the subgoals can eventually be erased down a path in the derivation tree, then an answer has been found (or a 'yes' answer computed). At this point the bindings for the variables can be used to give an answer to the original query.

unification of Prolog terms

Prolog unification matches two Prolog terms T1 and T2 by finding a substitution of variables mapping M such that if M is applied T1 and M is applied to T2 then the results are equal.

For example, Prolog uses unification in order to satisfy equations (`T1=T2`) ...

```
?- p(X,f(Y),a) = p(a,f(a),Y).
X = a  Y = a

?- p(X,f(Y),a) = p(a,f(b),Y).
No
```

In the first case the successful substituton is {X/a, Y/b}, and for the second example there is no substitution that would result in equal terms. In some cases the unification does not bind variables to ground terms but result in variables sharing references ...

```
?- p(X,f(Y),a) = p(Z,f(b),a).
X = _G182   Y = b   Z = _G182
```

In this case the unifying substitution is {X/_G182, Y/b, Z/_G182}, and X and Z share reference, as can be illustrated by the next goal ...

```
?- p(X,f(Y),a) = p(Z,f(b),a), X is d.
X = d   Y = b   Z = d
```

{X/_G182, Y/b, Z/_G182} was the most *general unifying substitution* for the previous goal, and the instance {X/d, Y/b, Z/d} is specialized to satisfy the last goal.

Prolog does not perform an *occurs check* when binding a variable to another term, in case the other term might also contain the variable. For example (SWI-Prolog) ...

```
?- X=f(X).
X = f(**)
```

The circular reference is flagged (**) in this example, but the goal does succeed {X/f(f(f(...)))}. However ...

```
?- X=f(X), X=a.
No
```

The circular reference is checked by the binding, so the goal fails. "a canNOT be unified with f(_Anything)" ...

```
?- a \=f(_).
Yes
```

Some Prologs have an occurs-check version of unification available for use. For example, in SWI-Prolog ...

```
?- unify_with_occurs_check(X,f(X)).
No
```

The Prolog goal satisfation algoritm, which attempts to unify the current goal with the head of a program clause, uses an instance form of the clause which does not share any of the variables in the goal. Thus the occurs-check is not needed for that.

The only possibility for an *occurs-check error* will arise from the processing of Prolog terms (in user programs) that have unintended circular reference of variables which the programmer believes should lead to failed goals when they occur . Some Prologs might succeed on these circular bindings, some might fail, others may actually continue to record the bindings in an infinite loop, and thus generate a run-time error (out of memory). These rare situations need careful programming.

Example 1: on Family Tree

The most common example of a logic program is the creation of a KBS of your family tree. Firstly, create a set of facts as follows using the 3 predicates:

male = { The males in your family. } – This is an unary predicate
female = { The females in your family. } – This is an unary predicate
parent_of = { The pairs (x,y) where x is the parent of y in your family. } This is a binary relationship between x and y, expressing the parenthood of x over y.

You need one entry for each person in your family. This forms the facts of the KBS. Using these three base predicates define the following predicates. (Note: avoid circular definition.) What are these subsequent set of relationship definitions called?

father, mother, son, daughter, grandfather,
aunt, uncle, cousin, ancestor, spouse

Possible example:

1. male(jerry).
2. male(stuart).
3. male(warren).
4. female(kathe).
5. female(maryalice).
6. brother(jerry,stuart).
7. brother(jerry,kathe).
8. sister(kather,jerry).
9. parent_of(warren,jerry).
10. parent_of(maryalice,jerry).

Using the definitions you have created, make a query ?- spouse(x, y).

a) Do a trace of the matching process in the AND-OR tree to show the search process.

b) Reorder the facts in the following way: 10, 9, 4, 5, 1, 2, 3, 8, 7, 6. repeat the query and do a trace on the new query. Are query results the same? Are the trace identical. If not, explain why.