HOW TO RUN PROLOG
Principles of Programming Languages 198:314
Spring 1999

We will be running Quintus Prolog on the undergraduate SUN cluster. The preferred way to run Prolog
is from within emacs, as then your output is in an emacs window and can be manipulated as text after you
exit the interpreter. However, you are not required to use the emacs environment for this project. In order
to use the on-line manual in emacs, execute the following command once, the first time you run Prolog:
cat ~/uli/cs314/prolog/quintus >> .emacs
This will append the file ~/uli/cs314/prolog/quintus onto your .emacs file, which is input whenever your
startup emacs.

We prefer to use a two window (i.e., buffer) emacs environment, which encourages an interactive style
of editing and running Prolog programs. The upper window is used primarily for editing of facts and rules.
The lower window (which is running the shell) is used for executing programs; it keeps a running record of
your execution. Thus, you can save the results of a Prolog session by saving this lower window using the
usual emacs file saving commands (e.g., Control-x Control-w).

TO START: When you have logged on, the command emacs will get you into GnuEmacs. Then, the
following command sequence should be executed (where Control-a stands for depressing the control key
while hitting the keyboard key a and Esc stands for hitting the escape key by itself). Note, these commands
are all GnuEmacs commands to setup your two windows for maximal ease of execution; the Prolog commands
are explained in the section INSIDE PROLOG.

Control-x 2 creates two windows (Emacs buffers) on your screen, one above the other.

Control-x o puts the cursor in the lower window; this is a toggle instruction which can switch you from
window to window and back again.

Esc x run-prolog this gets Quintus Prolog running in the lower window and sets up the upper window
for editing Prolog programs. You will see the Quintus Prolog header appear on the top of the lower
window followed by the Prolog prompt ?-.

After you execute this command, you type into the emacs buffer exactly as you would at the system
level. After you end the Prolog session by typing "halt!", you can save your output buffer to a file.

INSIDE PROLOG: You are now ready to start defining Prolog programs and running them. You can
save your Prolog programs and run them again; the expected Prolog source file extension is .pl. All rules
and facts with the same left-hand side predicate must be defined on the same file.

We expect you to run as follows:

1. Create your clauses in the emacs window and save them to a file foo.pl.
2. While running Prolog within emacs, use the load command to enter this file into the interpreter
database.
3. Then run some queries using the predicates you have defined, and test your clauses. You can make
changes easily within the emacs window, save them to foo.pl, reload the file into Prolog and then test
the changed version.

The following line load command is typed in at the Prolog prompt:
[filename].—“consults” (reads) the file named filename for input to the data base; assumes that the file
extension is .pl. Other files can be consulted by surrounding their full filename (with extension) in quotes.

Infinite loops in Prolog programs, usually caused by poorly written recursive rules, are easy to notice. If
the interpreter produces no output to the screen for a while, your program is probably in an infinite loop.
When running Prolog, Control-c Control-c (yes, it’s two of them) interrupts execution and results in the
Prolog prompt for help. If you reply to the prompt h, thus asking for help, these are some of the choices
open to you:
c continue - this resumes execution where you left off.

trace - debugger will be activated and will start creeping; this means a step-by-step trace of your execution is provided (see below).

d debug - debugger will be activated and will start leaping; this is a skipping trace that doesn’t trace every statement.

a abort - causes a Prolog execution abort; this gets you a prompt for more queries to be entered.

e exit - causes an irreversible exit from Prolog.

h help - prints a longer version of this list

To restart execution after an exit, you will have to use the Esc x run-prolog sequence again. Remember to use the emacs cut commands to remove the record of the former execution before beginning again.

The entire set of Quintus Prolog reference manuals is online and available on the undergrad SUns. It may be accessed by using the help command in prolog. For example, to find out about the prolog command print, type:

| ?- help(print).

You will then be in a help menu (if you are running prolog under emacs). The commands to access the menu will be listed on the bottom line of your emacs screen.

To leave Prolog and save the lower window, use the usual emacs file saving commands (e.g., Control-x Control-w). Then exit emacs as usual (e.g., Control-x Control-c). You will be asked about saving any buffers you have opened during the emacs session, as usual.

DEBUGGING AND TRACING: To trace a running Prolog program there are many options. You can trace the step-by-step execution of the program. You can trace the program, but skip some of the intermediate steps. You can watch just one predicate as the rules defining it are executed; this means execution proceeds as normal until that predicate is hit and then step-by-step tracing begins and continues as long as a rule for that predicate is executing.

trace. — starts step-by-step execution of subsequent queries; The trace command normally stops at every line; you must type carriage return to make it continue. Once you are in trace mode you can switch back to no trace mode by saying notrace, at the Prolog prompt.

While tracing, to get a step-by-step trace that doesn’t stop at each step and ask you for a carriage return, you use the leash command. The purpose of leashing is to allow you to speed up single-stepping (creeping) through a program by telling the debugger that it does not always need to wait for user interaction at every port. The leashing mode only applies to procedures that do not have spypoints on them.

To get back to a normal trace which stops after every step, type leash(all), at the Prolog prompt.

spy((clause-name)). — causes a trace of the execution of the specified clause only. During execution, once that clause is reached, the trace is presented and several options are possible:

Return — single step trace

1 — leap trace (allows you to skip over subgoals)

? — help

g — shows the ancestor list of current goals not yet satisfied

The following program was run on remus and traced as shown below. The annotations which follow the % on the right, were inserted after the run of the program.
Program Used (in file mem)

%these clauses test the first argument
memberof(A, [A | B]). %for membership in the list second argument.
memberof(A, [B | C]) :- memberof(A, C).

memberof(A, [A | _]). %these clauses do the same job as the above
memberof(A, [X | C]) :- memberof(A, C). %clauses but use the ‘don’t care’ variable.

Output Obtained in emacs:

Quintus Prolog Release 3.3 (Sun 4, SunOS 5.5.1)
Originally developed by Quintus Corporation, USA.
Copyright (C) 1996, AI International Ltd. All rights reserved.
Castle Chambers, High Street, Berkhamsted, Herts, UK. +44 (0)1442 873673
Email: support@ail.co.uk WWW: http://www.ail.co.uk
Licensed to LCSR/CF Rutgers University

| ?- [mem]. %reading file "mem.pl"
% compiling file /ug/u0/uli/cs314/prolog/mem.pl
* Singleton variables, clause 1 of memberof/2: B
* Approximate line: 1, file: ’/ug/u0/uli/cs314/prolog/mem.pl’
* Singleton variables, clause 2 of memberof/2: B
* Approximate line: 2, file: ’/ug/u0/uli/cs314/prolog/mem.pl’
% mem.pl compiled in module user, 0.010 sec 480 bytes

yes

% The debugger will first creep -- showing everything (trace)

yes
[trace]
| ?- memberof(a,[a,b]). %using clauses to check membership
   (1) 0 Call: memberof(a,[a,b]) ? %successfully
   (1) 1 Head [1->2]: memberof(a,[a,b]) ?
   (1) 0 Exit: memberof(a,[a,b]) ?

yes
[trace]
| ?- memberof(c,[a,b]). %using clauses to check membership
   (1) 0 Call: memberof(c,[a,b]) ? %unsuccessfully
   (1) 1 Head [1->2]: memberof(c,[a,b]) ?
   (1) 1 Head [2]: memberof(c,[a,b]) ?
   (2) 1 Call: memberof(c,[b]) ?
   (2) 2 Head [1->2]: memberof(c,[b]) ?
   (2) 2 Head [2]: memberof(c,[b]) ?
   (3) 2 Call: memberof(c,[]) ? %at lowest level of recursion has
   (3) 3 Head [1->2]: memberof(c,[]) ? %examined whole list and not
   (3) 3 Head [2]: memberof(c,[]) ? %found c
   (3) 2 Fail: memberof(c,[]) ?
   (2) 1 Fail: memberof(c,[b]) ?
(1) 0 Fail: memberof(c,[a,b]) ?

no
[trace]
| ?- notrace.
% The debugger is switched off

yes
| ?- mem(X,[a,b]). %using clauses to generate members
%of the list
X = a ;
X = b ;

no
| ?- trace.
% The debugger will first creep -- showing everything (trace)

yes
[trace]
| ?- mem(Z,[a,b]).
   (1) 0 Call: mem(_,6595,[a,b]) ?initial call of recursive clause
     (1) 1 Head [1->2]: mem(_,6595,[a,b]) ? %unification with 1st rule
       (1) 0 Exit: mem(a,[a,b]) ? %successful return with A bound
%to a
Z = a ; %forced reexecution by ":;"
   (1) 0 Redo: mem(a,[a,b]) ? %try 2nd rule in 1st level recursion
     (1) 1 Head [2]: mem(_,6595,[a,b]) ?
     (2) 1 Call: mem(_,6595,[b]) ?
     (2) 2 Head [1->2]: mem(_,6595,[b]) ? %use 1st rule in 2nd level recursion
       (2) 1 Exit: mem(b,[b]) ? %successful return with A
     (1) 0 Exit: mem(b,[a,b]) ? %bound to b
Z = b ; %forced reexecution by ":;"
   (1) 0 Redo: mem(b,[a,b]) ? %try 2nd rule in 2nd level recursion
     (2) 1 Redo: mem(b,[b]) ?
     (2) 2 Head [2]: mem(_,6595,[b]) ?
     (3) 2 Call: mem(_,6595,[]) ?
     (3) 3 Head [1->2]: mem(_,6595,[]) ? %can’t unify A with []
     (3) 3 Head [2]: mem(_,6595,[]) ?
     (3) 2 Fail: mem(_,6595,[]) ? %fail at 3rd level recursion
       (2) 1 Fail: mem(_,6595,[b]) ? %fail at 2nd level recursion
       (1) 0 Fail: mem(_,6595,[a,b]) ? %fail at 1st level recursion

no %report failure to user
[trace]
| ?- halt.

Process prolog finished