An Example Prolog Program

Introduction

When I started learning Prolog, I thought that the interactive console was great for trying things out and running queries real time, but I noticed an overall lack of instruction out there on writing a menu driven program that will take input. Because of that, I created a quick little program that will demonstrate a simple menu and some I/O.

The Knowledge Base

Let's get started with a quick knowledge base. This KB comes directly from page 42 of "Thinking as Computation" by Hector Levesque. There's nothing special here. It's just a basic KB that we can start with.

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

A Menu-based Driver

Now let's write a driver. Because the plan is to compile this into an executable, I'll call this main.pl. The first thing to do is to load your KB. This is accomplished with the following line:

```
:- [family].
```

Notice that the line starts with a :- and ends with a period. Now let's skip to the "main" function.

Main is what we're going to use to actually print the menu, so we'll start with a normal predicate definition and then we're just going to print out whatever we want/need for the menu. For this first simple example, we'll just give the user the choices to run a query or quit. At the end, we'll call read/1 to read in a variable (aptly named Choice) and then we'll call a predicate called run_opt and pass in whatever choice is. Finally, we call main again so that we keep displaying the menu until we're done.

```
main :-
    nl,
    write('>   Enter a selection followed by a period.'), nl,
    write('>   1. Run a query'), nl,
    write('>   2. Exit'), nl, nl,
    read(Choice),
    run_opt(Choice), main.
```

And finally we have the predicates that actually do the work. Here we want to run a query that the user inputs. That's going to involve another prompt and another call to read/1 to read in the query. We'll also call a couple of predicates that we haven't defined yet to print out the results of our query. We'll get to those definitions shortly.

```
run_opt(1) :-
    write('>   Enter a query followed by a period.'), nl,
    read(Query),
    print_query_true(Query),
    print_query_false(Query).
```

Notice that the first definition just has the number 1 as the parameter. That's because 1 was the menu option for running a query. When a 1 is selected, this definition of run_opt will be called. So now let's define a run_opt for menu option 2 and a run_opt for any other input.

```
run_opt(2) :- write('Goodbye'), nl, halt.

run_opt(_) :- write('Invalid option'), nl, halt.
```

And we're almost done! The last thing we need to do is define those predicates for printing. We want to print out the results of our query - true or false - so we'll define a predicate for each of them. The actual work isn't too bad. We'll use the forall/2 predicate to evaluate the query and print the result along with the word true or false as is appropriate.

```
print_query_true(Q) :-
    forall(Q, writeln(true:Q)).

print_query_false(Q) :-
    forall(\+ Q, writeln(false:Q)).
```

Note that we're doing basically the same thing but to print the false results, we are checking for a negative evaluation of our query.

And that's it. So put all together our main.pl looks like this.

```
%main.pl

:- [family].

print_query_true(Q) :-
    forall(Q, writeln(true:Q)).

print_query_false(Q) :-
    forall(\+ Q, writeln(false:Q)).

run_opt(1) :-
    write('>   Enter a query followed by a period.'), nl,
    read(Query),
    print_query_true(Query),
    print_query_false(Query).

run_opt(2) :- write('Goodbye'), nl, halt.

run_opt(_) :- write('Invalid option'), nl, halt.

main :-
    nl,
    write('>   Enter a selection followed by a period.'), nl,
    write('>   1. Run a query'), nl,
    write('>   2. Exit'), nl, nl,
```

```
    read(Choice),
    run_opt(Choice), main.
```

Compiling

Swi-prolog allows you to compile using the same command that you use tostart up the interactive interpreter. A full listing of the options is available online at swi-prolog.org. Swi-prolog.org is also kind enough to give a tutorial. The ones that I use are:

- --goal: This is basically to help prolog determine the exit code of the program. I'm using the output from the main predicate. This also clues prolog in on what predicate to start with so it will run main. Without this, when your program runs you'll get a prolog prompt and you will have to type in main with is not what we want.
- --stand-alone: This tells the compiler that this is a stand-alone application. I haven't noticed any problems if I fail to include this, but I include it anyway.
- --quiet: When prolog compiles it lists a whole bunch of information about what it is doing. Most of the time, I prefer not to see all of that, so I use --quiet although you might want to see that information especially as you start out.
- -o: This specifies the output file. Much like gcc, you get a.out if you do not specify a filename.
- -c: This is how you specify the source file. Note that in our example we can just use main.pl and it will pick up family.pl.

Putting these options together yields:

```
swipl --goal=main --stand_alone=true --quiet -o family -c main.pl
```

The Makefile

Because I don't like doing rework and because it makes for easier grading, I recommend/will require a makefile to compile your driver and your knowledge base into an executable. Notice that I don't have to include family.pl. I get that for free because it's included in main.pl. If you want a good tutorial on makefiles, I recommend you look at this one first. Here is the makefile for our example program.

```
CC=swipl
FLAGS=--goal=main --stand_alone=true --quiet
EXE=family
FILE=main.pl

$(EXE): $(FILE)
        $(CC) $(FLAGS) -o $@ -c $^

clean:
        -@$(RM) $(EXE) 2>/dev/null
```

Going further

The nice thing about this program is it takes any query in as input and will run it so your queries don't even have to be canned. If you wanted to make this more interesting, there are a couple of things you can try.

Canned Queries

Let's say I want or need some canned queries and I need to print out all of the males or all of the females in the example. Some code to accomplish that would look like this.

```
write_male(X) :- write(X), write(' is male.'), nl.
print_males :- forall(male(X), write_male(X)).

write_female(X) :- write(X), write(' is female.'), nl.
print_females :- forall(female(X), write_female(X)).
```

Notice that this prints out all of the returned values for X, not just the first one.

More Menu Options

Add menu options simply by adding additional definitions of run_opt where each hard coded number corresponds to a menu option so here menu option 1 would be print all males. Option 2 would be print all females.

```
run_opt(1) :- print_males.

run_opt(2) :- print_females.

run_opt(3) :-
    write(    'Enter a query.'), nl,
    read(Query),
    print_query_true(Query),
    print_query_false(Query).

run_opt(4) :- write('Goodbye'), nl, halt.

run_opt(_) :- write('Invalid option'), nl, halt.
```