COSC 4P79 Expert Systems Project

B. Ross, Winter 2012

Due date: End of term (TBA).

Objectives: An opportunity to apply the shell programming and knowledge engineering principles discussed during the course.

Hand in: (i) A 7-8 page report describing your system: This should include a description of the domain, a discussion of the system implementation, and a drawing of the system architecture. (ii) A complete source listing, and some example dialogs exercising main components of the system. (iii) Copy of all source code and executables required to execute your system onto a CD or DVD, with instructions on how to execute.

Group projects: You can work alone or in pairs for this project. If you work with someone else, include a listing of what tasks were undertaken by each group member.

System requirements:

1. **Domain**: This expert system is necessarily a toy system. However, you should pick a domain of expertise that is sufficiently complex so that you can implement a non-trivial system. You are not required to use a real "expert"; however, doing so would add substantially to the authenticity of the system. When choosing a domain, one thing to consider is that you and your acquaintances are probably experts in many areas. Although it isn't ideal for a knowledge engineer to play the role of the expert, you can do this for the project if necessary. There is much scope for the selection of a creative problem domain for an expert system. You might consider talking to faculty in other departments as potential experts for a problem in their domain area. If you are having problems deciding on a topic, two possibilities are as follows.

(i) *Course Advisor*. An expert system to determine which course a computer science major should take. This expert system encodes the knowledge taken from the university course calendar. The knowledge would primarily encode the different courses given by the department, and the relationships that courses have for one another (pre-requisite, co-requisite, terminal courses, year in degree ...). The system would need to prompt the user for information such as their intended degree, their year in the program, and the courses they have taken. The advice given would be in the form of suggested computer science courses to take, and why they are being advised (are they prerequisites to others? major or minor requirements? electives?).

(ii) *Automobile driver*: an expert system that can drive a car. This is not as trivial as you might expect! For example, there are many factors involved in driving *safely* through intersections: stop signs or traffic lights; the number of roads leading into the intersection; whether roads are one way or two way; the number of lanes in each road; what cars are where on the intersection; are there

malfunctioning lights, stalled cars, or other exceptional circumstances; and (most importantly) where does the driver wish to drive?

2. **Shell**: The system is to be implemented in Prolog. The main requirements of the system are as follows:

- The knowledge base and shell utilities should be clearly delineated and modularized. The knowledge base must be declarative. The shell programming should be as clean as possible.

- The system should prompt the user with clear messages. There should be a measure of input checking. The system shouldn't crash if miss-spelled data is entered. Menus are recommended.

- The system should incorporate an explanation facility appropriate to the style of inference used. For backward chaining, the user can enter "why" at any point, and the system should give an explanation of why a particular question is being asked. The explanation should be hierarchical: repeated "why" queries should result in higher-level explanations. Note that forward-chaining explanation is more limited than with backward chaining.

- The shell utilities should be properly commented with inline documentation. Explain how each predicate in your program is to be called and how it functions. A suggested format is the following:

```
/*  member(Item, List):

        input/output: constant Item
        input: list List

    Member is satisfied if Item is found in List.  Member is
    nondeterministic --        multiple solutions can be returned.   */
```

Comments:

The above is a minimal requirement of the system. You can get full marks if the above is implemented. However, you are welcome to implement any type of expert system shell that you wish -- backward chaining, forward chaining, probabilistic ... Enhancements to the above are encouraged.

Sicstus Prolog has a TCL/TK interface. You may wish to use it to create a GUI for your system.