Assignment 4

CSG120, Fall 2003
Due: Thursday, Nov. 6

Part I. Pencil-and-paper problems

1. Translate the two English sentences listed below into predicate calculus sentences in explicit quantifier form The only predicates you should use are these five:

```
computer        new        person        owns        programs
```

Your answers may use either Lisp-style or mathematical syntax, whichever you prefer. The meaning of these predicates is as follows:

```
computer(x)   or  (computer x)    means    "x is a computer"
new(x)        or  (new x)         means    "x is new"
person(x)     or  (person x)      means    "x is a person"
owns(x,y)     or  (owns x y)      means    "x owns y"
programs(x,y) or  (programs x y)  means    "x programs y"
```

The only constant you should use is john.

To get you started, the Lisp-style translation of ”John owns a new computer” should begin (exists (x) ...

a. ”John owns a new computer.”

b. ”Everybody who owns a new computer programs it.”

2. Do Exercise 9.18 in the textbook (p. 318) on resolution. To get you started, here is the solution to part (a) using lisp-style syntax:

```
Premise: (forall (x) (if (horse x) (animal x)))
Conclusion: (forall (x) (if (exists (y) (and (headof x y) (horse y)))
                            (exists (z) (and (headof x z) (animal z)))))
```

(To understand this, think of the conclusion as being ”Any head of any horse is the head of some animal.”)

Part II. Simple backward chainer that explains its reasoning

3. Write a general backward chaining program for definite clause predicate calculus sentences having no variables that “explains” its reasoning in an informative way, as described below. Call your program *explain-why* and assume that the knowledge base (which is fixed at initialization) is stored in a global variable (named *kb*, say). A call to your program will then look like (explain-why ’(eats john pizza)).

4. Test your program using the abstract knowledge base and queries that appear in the file "/programs/logic/hw4-data.lisp" accessible from the course web page.

Turn in hardcopy of your source files along with a dribble file showing proper behavior of your backward chaining program on the sample data.

The main challenge of this assignment is to produce an explanation of the program's reasoning when the conclusion is positive. For example (explain-why '(eats john pizza)) might print:

```
a. It is not true that (eats john pizza)

   or

b. It is known that (eats john pizza)

   or

c. It is concluded that (eats john pizza) because
  .
  .
  .
```

The answer in case b indicates that the fact (eats john pizza) was present in the knowledge base, while the answer in case c indicates that it had to be deduced, and the ". . ." stands for a "trace" of the program's steps in reaching the conclusion. Only the steps leading to the first successful solution found need to be shown.

For example, suppose the knowledge base contains the sentences:

```
  (eats john spaghetti)
  (if (eats john spaghetti) (eats john pizza))
```

Then (explain-why '(eats john pizza)) might yield the output:

```
It is concluded that (eats john pizza) because
  it is known that (eats john spaghetti)
```

Another example: Suppose the knowledge base contains the sentences:

```
  (eats john spaghetti)
  (eats mary pizza)
  (if (and (eats john spaghetti) (eats mary pizza))
      (eats john pizza))
```

Then (explain-why '(eats john pizza)) might yield the output:

```
It is concluded that (eats john pizza) because
  it is known that (eats john spaghetti) and
  it is known that (eats mary pizza)
```

Note the use of indentation to help organize the explanation. This is especially helpful if the explanation is based on several levels of backward chaining (e.g., if (eats mary pizza) itself had to be deduced from the knowledge base in this last example). For full credit your program must use indentation or some other equally clear method of indicating this kind of nested explanation structure.

HINT 1: As you search for a solution, build a list of "explanatory" elements. Then, once you have a solution, you can use this list to produce the explanation. Note that this list corresponds essentially to the solution path returned by the search routines considered earlier in the course. Just as in the earlier assignments involving search, your program should return only the first solution path found.

HINT 2: Note that you will not have to use any unification or variable substitution in your program; you only need to check for exact matches. In order to use backward chaining, however, you still have to select a search strategy for deciding which subgoals to explore next, and the simplest to program is a depth-first, left-to-right strategy, as discussed in class. Furthermore, you may omit all repeated-node checking if you wish, since the sample data you are to use will not lead to any infinite search paths. (This also means that you need not impose any depth limit.)