Assignment 7:
Prolog Programming

15-317: Constructive Logic

Out: Thursday 10/24/13
Due: Thursday 10/31/13 before class
Total points: 45

The purpose of this assignment is to familiarize you with the basics of programming in (nearly) pure Prolog, and to get you to experiment with the behavior of backtracking search on the deductive systems we studied in the first part of the semester.

Submit your files before the beginning of class by copying them to the directory

/afs/andrew/course/15/317/submit/<userid>/hw07

where <userid> is replaced with your Andrew ID.

1. Prolog programming

For this part of the assignment, you will write and test some simple Prolog predicates, adhering to the following standards:

- Don’t use cut, conditional, or any other control operator.
- Don’t use negation-as-failure or disequality.
- You may use any of the list processing predicates or arithmetic predicates provided by gprolog, except maplist or any of the sorts.
- Backtracking must not result in nontermination or in wrong answers. That is, when the user types ; in response to a query result, execution must terminate with a correct answer or no.
- Backtracking may produce redundant correct answers.
- When your code is loaded and compiled, no warnings should be printed.
- Each predicate definition must be preceded by a comment specifying the intended modes for all arguments: + for input, - for output, or ? for both. Example: % member1(+X,-Xs) succeeds if X is in the list Xs
- Put your definitions in one file named part1.pl, clearly delineating the parts with comments.

10 (a) Define the following operators for negation, conjunction, disjunction, and implication. In order of priority:

1. Negation neg (prefix)
2. Conjunction and (infix, left associative)
3. Disjunction or (infix, left associative)
4. Implication implies (infix, right associative)

Define predicate eval/2. The intended use is that the first argument is an input Boolean formula consisting of true, false, and logical operators, and the second is the result of evaluating that formula.

Example:

```
?- eval(true or a, V).
V = true ? ;
```

```
no
?- eval(true and a, V).
```

```
no
?- eval(true and false, V).
V = false
```

```
yes
```

(b) A propositional formula is in conjunctive normal form if it is a conjunction of clauses, where each clause is a disjunction of literals, and a literal is either an atom or the (single) negation of an atom.

Define the predicate cnf/1, which succeeds iff its argument is a formula in conjunctive normal form.

Example:

```
?- cnf(a and b or c).
```

```
no
?- cnf(a and (b or c)).
```

```
yes
?- 
```

(c) Polynomials can be represented in Prolog as lists of pairs (c, n). Each pair represents one term with coefficient c and exponent n.

Define a predicate poly_sum/3 to compute the sum of two polynomials represented in this way. The definition should not depend on any particular ordering of the lists, but its behavior in the presence of duplicate exponents in a list is undefined (you may do whatever you like in that case).

Example:

```
?- poly_sum([[1,5),(1,8),(1,0)],[[1,8),(2,0),(2,5),(2,1)],P).
```

```
P = [[3,5),(2,8),(3,0),(2,1)] ?
```

(d) For this exercise you will work with difference lists. As we did in class, you will use the infix operator \ which you should define as follows in your source code:
Define a predicate `shift/2` where both arguments are difference lists, and the second is formed by “shifting” the first argument left and adding the head element to the end. For example:

```
?- shift([a,b,c,A]\A,L).
```

\[ A = [a|B] \]
\[ L = [b,c,[a|B]]\backslash B \]

`yes`

Don’t use `append` (either its normal definition or the difference list version). However, it may be helpful to think about how your definition would work if you did use `append` (for difference lists).

2. For this question you will write straightforward translations of natural deduction and the sequent calculus for minimal logic (there is no rule for falsehood) for pure propositions (no quantifiers). Your code should adhere to the standards above, except that it’s not required to terminate, for obvious (we hope!) reasons.

Use the operators you defined above (`neg, and, or, implies`) to represent propositions. Put all the code for this part in a file named `part2.pl`.

(a) Define a predicate `prove/2`; `prove(Gamma,A)` should succeed iff there is a natural deduction proof of `Γ ⊢ A` that does not use falsehood elimination. E.g.

```
prove([], a implies a).
```

`true`

(b) Define a predicate `entails/2`; `entails(Gamma,A)` should succeed iff there is a sequent calculus proof of `Γ ⇒ A` that does not use the left rule for falsehood. E.g.

```
?- entails([a], a).
```

`true`

(c) In comments, give four different formulas for each predicate where the predicate succeeds. Also give a formula where `entails` succeeds but `prove` does not terminate, and explain why.