CSE 3401 Assignment 2
Winter 2013

Date out: February 12. Date due: March 3, at 11:30 pm

- The submitted assignment must be based on your individual work. Review the Academic Honesty Guidelines for more details.
- This assignment constitutes 6% of your total mark for the course and is marked out of 120.
- You should adhere to the coding guidelines posted on the website; comment your code and test it thoroughly.
- You may define any auxiliary relations if that helps in defining the required predicates.
- You should Submit 2 files for this assignment:
  - a2.pl, which is the source code of your solutions for questions 1 and 2. This file should follow the Prolog solutions format provided on the website (solutionFormat.txt). Also, include any references you may have used.
  - a2.txt, which consists of the test cases you have used to test the predicates in questions 1 and 2, as well as the results of testing. Include a header with your name, student number and cse login.

Soft Copies: Gather all the required files in a directory named `a2answers` and submit it electronically by the deadline. To submit electronically, use the following Prism lab command:

```
submit 3401 a2 a2answers
```

Alternatively, you may use web submit (https://webapp.cse.yorku.ca/submit/) and choose the correct course and assignment number to upload your files.

Question 1: Finite State Automata (65 Points)

This exercise builds on the discussion of deterministic finite-state automata in the slides Recursion, divide and conquer, text processing (2 -11).

In order to have several automata defined at the same time, let's give automata names. We can define a predicate `automaton` which relates a name to an automaton structure as a term:

```
automaton(name, [initialState,listOfFinalStates, transitionList]).
```

For example, the automaton described on slide 5 could be represented by asserting

```
automaton(evenXoddY, [ee, [eo], [
                  ee-x-oe, ee-y-eo,
                  oe-x-ee, oe-y-oo,
                  oo-x-eo, oo-y-oe,
                  eo-x-oo, eo-y-ee ]).
```

Q1.a. Rewrite the `fsa` and `scan` predicates so that `fsa` has as arguments the name of an automaton and an input sequence, and `scan` has three arguments: automaton name, input sequence, and current state. You can test it on the evenXoddY automaton defined above, for example:

```
?- fsa(evenXoddY, [x,x,y]).
True.
```

Q1.b. Demonstrate your automaton predicate by defining an automaton called `nice` with the structure of the Wikipedia example "Fsm parsing word nice". You can test it by providing test cases such as:

```
?- fsa(nice, [n,i,c,e]).
True.
```

Q1.c. Implement a Prolog predicate `empty(A)` which determines whether the language accepted by automaton A is the empty set. (Hint: the language is empty if no final state is reachable from the initial state. To facilitate testing, you need to define a new automaton called emptyTest. The language accepted by automaton emptyTest is the empty set). You can test the predicate by providing test cases such as:

```
?- empty(emptyTest).
True.
```

Q1.d. Implement the predicate `disjoint(FSA1, FSA2)` which tests whether the languages accepted by the automata `FSA1` and `FSA2` are disjoint, i. e., there is no input sequence which causes both FSAs to enter one of their final states. Assuming you have defined an automaton named good, which accepts "good", you may test the predicate by running test cases such as:

```
?- disjoint(nice, good).
True.
```

Q1.e Implement the predicate `infinite(A)` which tests whether the language accepted by the automaton A is infinite (Hint: you may start by looking for a loop in the state-table. To facilitate testing, you need to define a new automaton called infTest. The language accepted by infTest is infinite). You can test the predicate by providing test cases such as:

```
?- infinite(infTest).
True.
```

Question 2: Conversion to Conjunctive Normal Form (55 Points)

Write a program for converting a propositional formula into conjunctive normal form. The main predicate of this program, `normalize(PF, CNF)`, takes `PF` , which is a propositional formula as input from the user. This formula is converted to conjunctive normal form and `CNF` is instantiated with the result.

Proposition symbols can be any Prolog atoms, except for 'v' which will be defined as the 'or' operator. In addition to the propositional symbols, the formula may contain one or more of the following connectives: implication (=>), equivalence (<=>), negation (-), conjunction (`&`) and disjunction (v) . The operators can be declared as follows:

```
:- op(800, xfy, [&]). /* Conjunction */
:- op(850, xfy, [v]). /* Disjunction */
:- op(870, xfy, [=>]). /* Implication */
:- op(880,xfy, [<=>]). /* Equivalence */
```

For negation, you can use [-] which is a predefined operator in Prolog (a list of predefined operators is available at: http://www.swi-prolog.org/pldoc/doc_for?object=op/3).

It might be helpful to follow the same order of steps explained in the slides Inference in First-Order Logic while converting the formula to CNF. Note that since the `normalize(PF, CNF)` predicate accepts propositional formula as input, your program does not have to carry out the following steps: Standardizing Variables, Skolemizing and Converting to Prenix form.

You may test this predicate by running test cases such as:

```
?- normalize(- b => c & d , CNF).
CNF = ((b v c) & (b v d)).

?- normalize((q => e) & c, CNF).
CNF = ((-q v e) & c).
```