C120.2: Declarative Programming II: Prolog Main Test

This test comprises four compulsory parts **a** to **d**.

Execute the query `?- use_module(library(lists)).`
to gain access to the list-processing module.

Enter your programs into the file whose name will be specified to you.
Try to avoid using the cut (!) —it is not needed in this test.

Introduction

A particular form of propositional logic employs just the connectives ¬ and □. Formulas in this logic can be represented in Prolog by using

- an atom to represent an atomic proposition,
- a compound term neg(X) to represent a formula ¬(X),
- a compound term imp(X, Y) to represent a formula (X□ Y).

For example, imp(q, imp(imp(r, q), neg(p))) represents (q□ ((r□ q) □ ¬(p))).

The Test

a **Write a program** defining the predicate atprops(F, As) which, given F representing a formula, returns As as a list of the atomic propositions occurring in F. It doesn't matter how As is ordered or whether it contains duplicates. For instance, the query

```
?- atprops(imp(q, imp(imp(r, q), neg(p))), As).
```

might return As as [q, r, q, p] or as [p, q, r], depending how you program it..

Hint — write a recursive "atprops" clause for the case where F has the form imp(X, Y),
another recursive clause for the case where F has the form neg(X),
and a base case for the case where F is an atom.
Note that X and Y here may be either atomic or compound.

b **Write a program** defining the predicate interp(As, Int) which, given As as a list of atoms, returns Int as a list that associates a single logical value with each atom. Specifically, for each distinct A in As, Int must contain exactly one pair of the form (A, V), where V is either t or f. For instance, the query

```
?- interp([q, r, q, p], Int).
```

might return Int as [(p, t), (r, f), (q, f)], being one of 8 possible solutions. Ensure that your program can return all those solutions.

Hint — write the clause

```
interp(As, Int) :- interp1(As, [], Int).
```

and write a program for "interp1(As, Acc, Int)" that accumulates (A, V) pairs in Acc whilst ensuring that each A appears only once in Acc. The appropriate base case is

```
interp1([], Int, Int).
```

c **Write a program** defining the predicate satis(F, Int) which holds only in the case that, given F representing a formula and given Int as an interpretation, Int satisfies F —that is, F evaluates to t on the basis of the values that Int associates with the atomic propositions occurring in F. For instance, the query

```
?- satis(imp(q, imp(imp(r, q), neg(p))), [(p, t), (r, f), (q, f)]).
```

should succeed because (q → ((r → q) → ¬(p))) evaluates to t when q has value f.

d **Write a program** defining the predicate valid(F) which holds only in the case that given F represents a formula that is satisfied by all interpretations.

Hint — F is valid if it is not invalid. It is invalid if some interpretation fails to satisfy it. So just make appropriate use of your existing "atprops", "interp" and "satis" programs (and "\+") to test F on this basis.

Hints for Survival

Take care to *avoid trivial syntax errors* such as mismatched brackets; mis-spelled predicate symbols; illegal spaces between predicate symbols and left brackets; or the omission of **full-stops** from the ends of clauses or queries.

A commonly-seen compilation warning occurs if you have a clause containing a *named* variable that Sicstus thinks need not be named, as in these examples:

p(X, Y) :- q(Y).	Sictsus prefers ...	p(_, Y) :- q(Y).
p(X) :- q(X, Y).	Sicstus prefers ...	p(X) :- q(X, _).

So, always use an *underscore* when the variable's name is irrelevant to the logic.

Remember to *load the list-processing module*, as described in the test paper.

After entering or editing your code, always *resave and recompile* it before attempting new querying.

If your programs compile correctly but compute wrong results, debug them by inserting appropriate write calls enabling you to print out appropriate terms at intermediate stages in the execution.

Useful Prolog primitives include these, but you *need* only the first five:

member(X, Y)	X is a member of list Y
append(X, Y, Z)	appending list Y onto list X gives list Z
atom(X)	succeeds iff X is an atom
\+ P	succeeds iff the call-term P fails finitely, and fails finitely iff P succeeds (if P is not an atomic call, put a space after the \+)
X =.. L	converts between a term X and a list L whose head is the principal functor of X and whose tail comprises the arguments of that functor e.g. f(a, Z) =.. L returns L as [f, a, Z] e.g. X =.. [f, a, Z] returns X as f(a, Z)
length(X, L)	the length of list X is the number L
X \== Y	X and Y are not identical terms
findall(T, P, L)	returns L as the list of all instances of term T for which the call-term P can be solved; P may or may not refer to T; at the instant prior to calling, L must be unbound
forall(P, Q)	succeeds iff every way of solving P also solves Q; it succeeds even if P has no solutions; you can define it in Sicstus by entering the clause forall(P, Q) :- \+ (P, \+ Q).
write(X)	prints out the term X on the current output line
nl	throws a new line when printing