COMP 410
Spring 2018
Final Practice Exam

The topics on this practice exam reflect **ONLY** those which have been covered since the last exam. The real final is **CUMULATIVE**, so it will include questions similar to the previous practice exams. However, the final will be biased towards the sort of questions below.

Prolog Metainterpreters

1.) Write a metainterpreter which shows the number of conjunctions which were needed to compute a particular solution. Example queries follow. Your metainterpreter needs to handle only the rules necessary to execute these queries below.

```
?- interpret((X is 1 + 1, Y is 2 + 2), ConjunctionCount).
X = 2, Y = 4, ConjunctionCount = 1.

% This definition is used in the query below
% myLength([], 0).
% myLength([_|T], Len) :-
%    myLength(T, TLen),
%    Len is TLen + 1.

?- interpret(myLength([a, b, c, d], Len), ConjunctionCount).
Len = 4, ConjunctionCount = 4.
```

Second-Order Procedures

2.) Consider the following procedure:

```
foo(1, 2, 3, 4).
foo(1, 2, 4, 3).
foo(3, 4, 1, 2).
foo(1, 2, 1, 6).
```

With the above procedure in mind, provide answers to the following queries. You should provide solutions for all variables which have solutions (hint: this won't necessarily be all variables involved).

2.a) `?- findall(A, foo(A, B, C, D), E).`

2.b) `?- bagof(bar(C, D), foo(A, B, C, D), E).`

2.c) `?- bagof(bar(A, C, D), foo(A, B, C, D), E).`

2.d) `?- bagof(A, C^D^foo(A, B, C, D), E).`

2.e) `?- setof(A, B^D^foo(A, B, C, D), E).`

Mercury

3.) Write a procedure named `sublist` which will take a list and nondeterministically produce lists which contain contain elements in the input list. Example queries follow:

```
?- sublist([], List).
List = [].

?- sublist([1], List).
List = [1] ;
List = [].

?- sublist([1, 2], List).
List = [1, 2] ;
List = [1] ;
List = [2] ;
List = [].
```

Be sure to write appropriate `pred` and `mode` annotations. You only need to write mode annotations corresponding to the queries above.

4.) Write a procedure that conforms to the following `pred` and `mode` annotations:

```
:- pred foo(int, int).
:- mode foo(in, out) is multi.
:- mode foo(in, in) is semidet.
```

5.) Consider the following Mercury code, which does not compile as written:

```
:- type my_type(A) ---> foo(A) ; bar(A, A) ; baz(A, A, A).

:- pred something(my_type(A), A).
:- mode something(in, out) is det.
something(foo(A), A).
something(bar(A, _), A).
```

5.a) Why doesn't this code compile?

5.b) It is possible to get this code to compile by changing the `type` definition. Write a revised `type` definition below which will allow this code to compile. You may assume that this code is in complete isolation, so changing the `type` definition won't break anything elsewhere.

5.c) It is possible to get this code to compile by changing the `mode` annotation. Write a revised mode annotation below which will allow this code to compile. As before, you may assume the code is in complete isolation.

5.d) It is possible to get this code to compile by adding another line of code at the end. Write this added line below. This line may do whatever it wants, as long as it allows the code to compile. As before, you may assume the code is in complete isolation.