Spring, 1994 Name: ________________________

Com S 541 — Programming Languages 1

Test on λProlog, Logic Programming, Type Checking, and Type Inference

Special Directions for this Test

This test has 5 questions and pages numbered 1 through 5.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Unless we specifically request, your code does not have to run "backwards". To save space in the examples we do not print the input to tell the system to stop backtracking (a newline) and the system's response of "yes".
You can use cut (`!`) and `not` in your code.

General Directions from the "Course Specification"

Short answer questions will be graded on the basis of completeness (did you list all the relevant parts of the answer?) soundness (did you make any errors of fact or contradict yourself?) and clarity. You can help ensure clarity by giving examples where possible.
For essays, critical justifications, and the like there may be no one right answer; so the criteria for judgement will be whether your essay is:

- clear (you use examples where appropriate, you write clear sentences, diagrams, and you avoid excess verbosity),
- sound (you start from facts or reasonable assumptions, your logic is convincing), and
- complete (you consider relevant aspects of the problem and alternatives).

The highest grade will be earned by an answer that is, in addition, concise and elegant; that is, answers that say the most with the least amount of excess. Extra consideration will be given to answers that are especially creative. For programming problems, there is usually more than one right answer, hence it is important to present with your program evidence that it is correct. Answers to programming questions will be scored on the following points:

- elegance and clarity (conciseness, using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution, and how clear and convincing your argument is that the code is correct)
- correctness (whether your program solves the given problem correctly in the given programming language)

You should try to eliminate all syntax errors from your programs, as they will affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Run-time space and time efficiency of programs is not important, unless the problem says so. Also, checking of external inputs, prompting, labeling output, and other conventions for dealing with humans are not important (for this class), unless the problem says so.
The breakdown of points between these categories is roughly as follows.

- 50% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 50% for correctness (whether your program solves the given problem correctly in the given subset of the given language)

You are encouraged to use auxiliary subroutines whenever they are helpful.

1. (10 points) What is the (most general) type of SML function `fl` defined by the following? (If it has no type, explain what the type error is, by pointing out any subexpressions that have type errors. If it has a type, you don't have to give a proof, just write down what it is.)

```
fun fl f b [] = b
  | fl f b (x::ls) = fl f (f(b,x)) ls;
```

2. (10 points) Give (a) an example of a unifier, and (b) explain the importance of unifiers in logic programming. (Be sure your notation for part (a) is either what we used in class, or something that you explain clearly.)

3. (18 points) In this problem, you'll say what λProlog would print in response to a query after loading the following two modules. (Note that the first is the `signature` module from homework 7.)

```
module signature.
kind sig        type -> type.
type empty_signature    (sig T).
type add_type           T -> (sig T) -> (sig T).
type add_message        string -> (list T) -> T -> (sig T) -> (sig T).
type resType            (sig T) -> string -> (list T) -> T -> o.
type is_type            (sig T) -> T -> o.

resType (add_type Typ Sigma) MName Args ResultType :-
        resType Sigma  MName Args ResultType.
resType (add_message MName Args ResultType Sigma ) MName Args ResultType :- !.
resType (add_message MName' Args' ResultType' Sigma) MName Args ResultType :-
        resType Sigma MName Args ResultType.

is_type (add_type Typ Sigma) Typ :- !.
is_type (add_type Typ' Sigma) Typ :- is_type Sigma Typ.
is_type (add_message MName Args ResultType Sigma) Typ :- is_type Sigma Typ.


module disease.
import signature.
type diseaseSig (sig string) -> o.

diseaseSig S :-
    S = (add_type "Human"
        (add_type "Germ"
        (add_type "Void"
        (add_message "cough" ("Human"::nil) "Germ"
        (add_message "infect" ("Germ"::"Human"::nil) "Void"
        (add_message "sicken" ("Human"::nil) "Void"
        (add_message "die" ("Human"::nil) "Void"
        (add_message "die" ("Germ"::nil) "Void"
        (add_message "regain_health" ("Human"::nil) "Void"
        empty_signature))))))))).
```

Fill in the blanks in the output below to show what λProlog would print for the following query.

```
?- use "signature".
?- use "disease".
?- diseaseSig S,
   is_type S Answer_A,
   resType S B C D,
   resType S "die" ("Human"::E) F.

Answer_A = __________________,
B = _________________________,
C = _________________________,
D = _________________________,
E = _________________________,
F = _________________________,
S = (add_type ...) % you don't have to write anything for S
```

4. (10 points) Assuming that type checking rules for other statements and expressions are already given, give a type checking rule for a `while` statement (as in Pascal) in the λProlog notation used in homework 7. That is, suppose you were to add a `while` statement to mini-Cecil, what would the type checking rule be in λProlog?

   You may assume that the abstract syntax of a `while` statement is given by the following.

   ```
   type whileStmt expression -> statement -> statement.
   ```

5. (20 points) Consider the following module in λProlog.

```
module sets.
kind set  type -> type.
type empty      (set T).
type insert     T -> (set T) -> (set T).
```

What would you have to add to this module to implement a predicate **equal_as_set** which succeeds just when its two arguments have the same elements (in the usual mathematical sense)? For example, with your additions the following session would show how to use **equal_as_set**.

```
?- use "sets".
?- (equal_as_set empty (insert 3 empty)).
no
?- (equal_as_set empty empty).
solved
?- (equal_as_set (insert 4 (insert 3 empty)) (insert 3 (insert 4 empty))).
solved
?- (equal_as_set (insert 5 (insert 6 empty))
                 (insert 5 (insert 5 (insert 6 (insert 6 (insert 5 empty)))))).
solved
?- (equal_as_set (insert 5 (insert 5 (insert 6 (insert 6 (insert 5 empty)))))
                 (insert 5 (insert 6 empty))).
solved
```