Spring, 1995 Name: ________________

Com S 541 — Programming Languages 1

Test on λProlog and Logic Programming

Special Directions for this Test

This test has 5 questions and pages numbered 1 through 6.
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. (15 points) Write rule(s) in λProlog that define a higher-order predicate `foldleft` with the following type.

```
type   foldleft   (A -> B -> A) -> A -> (list B) -> B -> o.
```

The idea is that `foldleft` should be just like the functional studied in our unit on functional programming. That is, the second argument is used as the result in the base case, and in general the following holds.

```
foldleft F Init (X1::X2::...::Xn::nil) (F ... (F (F Init X1) X2) ... Xn).
```

2. (25 points) In this problem, you will write a λProlog module, `intTree`, that has the following parts.

   (a) It specifies a type for working with binary trees of integers, `intTree`. This type should be approximately like the SML declaration below. (It won't be exactly the same.)

   ```
   datatype intTree = Lf | Br of int * intTree * intTree;
   ```

   (b) It specifies a relation `incOver5`. This relation should have type

   ```
   type incOver5  intTree -> intTree -> o.
   ```

   that takes 2 arguments, and such that `incOver5 T1 T2` is satisfied when `T2` is exactly like `T1`, except that every label at which there occurs a number strictly greater than 5, that label is replaced with the old value plus one (1).

   Write the *complete* module below.

3. (10 points) Both SML and λProlog have a form of pattern matching for invocations. Briefly describe how they differ, and give an example that illustrates the difference.

4. This question deals with the two main interpretations of a λProlog program.

   (a) (5 points) Briefly describe a difference between the declarative (i.e., logical) interpretation of a λProlog program and its procedural interpretation.

   (b) (5 points) Give an example that illustrates the difference you described above.

   (c) (10 points) What practical difficulties would there be, if any, in making λProlog strictly obey the logical interpretation?

5. (20 points) The relational algebra is a feature of some relational database systems. In this problem you are to write a λProlog module that defines the following operators.

```
module relation_algebra.

type project_left  (A -> B -> o) -> (A -> o).
type project_right (A -> B -> o) -> (B -> o).
type restrict_left (A -> B -> o) -> (A -> o) -> (A -> B -> o).
type join          (A -> B -> o) -> (B -> C -> o) -> (A -> B -> C -> o).
type compose       (A -> B -> o) -> (B -> C -> o) -> (A -> C -> o).
```

To give an example, consider the following example database module.

```
module employee_example.

kind   skill  type.

type   accounting     skill.
type   programmer     skill.
type   logician       skill.
type   scientist      skill.
type   coffee_drinker skill.

type   employee           int -> string -> o.
type   expertise          string -> skill -> o.
type   names              string -> o.

employee 1 "I M Dull".
employee 2 "E G Hacker".
employee 3 "W R  Gates".
employee 4 "D L  Pigozzi".
employee 5 "K  Goedel".
employee 6 "A L  Einstein".
employee 7 "J  Lutz".

expertise "I M Dull" accounting.
expertise "E G Hacker" programmer.
expertise "W R  Gates" programmer.
expertise "W R  Gates" accounting.
expertise "D L  Pigozzi" logician.
expertise "K  Goedel" logician.
expertise "J  Lutz" scientist.
expertise "J  Lutz" coffee_drinker.

names "I M Dull".
names "W R  Gates".
names "J  Lutz".
```

The following gives some queries using this module.

```
?- use "relation_algebra".
?- use "employee_example".
?- (project_left expertise) N.
N = "I M Dull".
```

```
;
N = "E G Hacker".
;
N = "W R  Gates".
;
N = "W R  Gates".
;
N = "D L  Pigozzi".
;
N = "K  Goedel".
;
N = "J  Lutz".
;
N = "J  Lutz".
;
no more solutions
?- (project_right expertise) E.
E = accounting.
;
E = programmer.
;
E = programmer.
;
E = accounting.
;
E = logician.
;
E = logician.
;
E = scientist.
;
E = coffee_drinker.
;
no more solutions
?- (restrict_left expertise names) X Y.
Y = accounting,
X = "I M Dull".
;
Y = programmer,
X = "W R  Gates".
;
Y = accounting,
X = "W R  Gates".
;
Y = scientist,
X = "J  Lutz".
;
Y = coffee_drinker,
X = "J  Lutz".
;
no more solutions
?- (join employee expertise) V N E.
E = accounting,
N = "I M Dull",
```

```
V = 1.
;
E = programmer,
N = "E G Hacker",
V = 2.
;
E = programmer,
N = "W R  Gates",
V = 3.
;
E = accounting,
N = "W R  Gates",
V = 3.
;
E = logician,
N = "D L  Pigozzi",
V = 4.
;
E = logician,
N = "K  Goedel",
V = 5.
;
E = scientist,
N = "J  Lutz",
V = 7.
;
E = coffee_drinker,
N = "J  Lutz",
V = 7.
;
no more solutions
?- (compose employee expertise) N E.
E = accounting,
N = 1.
;
E = programmer,
N = 2.
;
E = programmer,
N = 3.
;
E = accounting,
N = 3.
;
E = logician,
N = 4.
;
E = logician,
N = 5.
;
E = scientist,
N = 7.
;
E = coffee_drinker,
```

```
N = 7.
;
no more solutions
```