Spring, 1995 Name: _____

Com S 541 — Programming Languages 1

Final Exam

Special Directions for this Test

This test has 6 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.

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. (5 points) Briefly answer the following question. In the computational semantics of the λ-calculus: how is a normal form defined?

2. (10 points) In the SML type inference system:
   (a) does the following expression have a type?

   ```
   (fn f => (fn x => (f (x x))))
   ```

   (b) Explain your answer.

3. (10 points) Is it possible to have completely static type checking in a language with dynamic scoping? Why or why not?

4. (10 points) What is the difference between the language of denotational semantics and SML? That is, if you tried to take a denotational semantics and translate it into SML, what kinds of changes would you have to make that would affect the meaning of the semantics? (Hint: think about *all* the parts of a denotational semantics.)

5. (15 points) (a) Is the writing of a formal semantics and a formal set of type checking or type inference rules a sensible thing to do when designing a programming language? Why or why not? (b) What limitations are there, if any, on the utility of these techniques?

6. Many specification languages have at their core the language of predicate logic with equality. The language of predicate logic involves no quantifiers, but only identifiers, `=` (equality), and the logical operators **not**, **and**, and **or**. Identifiers can stand for values or functions (functions that return a value of type `bool` are called predicates). To simplify the problem, we will only consider functions that take one (1) argument, such as `increment`.

   Here are some sample expressions in our limited version of predicate logic with equality (the examples are separated by commas): `i = j`, `b`, **not**`(b)`, **not**`(increment(i) = j)`, `(add2(k) = j)` **and** `(add3(j) = k)`, `(x = z)` **or** `(y` **and** `q = z)`, **(not**`(i = j))` **and** `(b` **and** `(q = z))`, `prime(i)`, **not**`(prime(j))` **and** `positive(k)`, `(add2(add3(k)) = j)`.

   (a) (10 points) Give an abstract syntax for expressions written in the language of predicate logic with equality. You don't need to give a syntax for identifiers or for declarations, but use identifiers in your syntax. Recall that the syntax only has to handle functions that take one argument. (If you can't get the grammar, you can buy the answer to this part for 10 points; ask during the test.)

   (b) (20 points) Give a set of type inference rules for your abstract syntax. Assume that identifiers appearing in an expression have been declared, and that there is more than one type. Your rules will be judged both on their form and their reasonableness.

(c) (20 points) Give an operational semantics (either a computational or evaluation semantics) for predicate logic with equality. You may assume that a semantic function, *equal*, is available that compares two values for equality. Functions when called either return a value or are undefined; you may assume there is a function *apply* that maps a pair, consisting of a function name and its argument value, to either a result value or $\perp$. For example, it might be that *apply*(`double`, 3) = 6, and *apply*(`divide4by`, 0) = $\perp$. (Don't use *apply* when giving the semantics of the logical operators.) Assume that both *equal* and *apply* are strict, in that when either argument is $\perp$, then their result is $\perp$.

Your semantics must make the logical operators **and** and **or** be evaluated using short-circuit evaluation; for example, if `i` and `k` are defined, then `(i = i)` **or** `(k = divide4by(zero))` is true, and **(not**`(k = k))` **and** `(i = divide4by(zero))` is false. However, the expression `(i = divide4by(zero))` **and** **(not**`(k = k))` is $\perp$ if the value of `divide4by(zero)` is $\perp$. Similarly, **not**`(i = divide4by(zero))` is $\perp$.