Spring, 1992 Name: ______________________

CS 342 — Principles of Programming Languages

Final Exam (Version 1)

Before you begin, please take a moment to look over the entire test so that you can budget your time.

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.

- 60% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 40% 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.

For this test you may use without further mention any code in the book from chapter 1, LISP, Scheme, CLU, Smalltalk, or Prolog. However, make sure that:

- When solving a problem in language X, the code you are using is written in the book in language X or was translated from an earlier chapter into language X in the library for language X. Functions that you may assume are available in every language (except Prolog) include: `+1`, `or`, `and`, `not`, `<>`, `<=`, `divides`, `mod`, `gcd`, and `abs`.

  That is, don't use code from language Y when solving a problem in language X.
- If you have any doubts about our being able to find the code you are using, you give us a page number where we can find the code. This is especially important if the chapter has multiple versions of a function with different names.

1 Scheme

1. (5 points) Consider the following Scheme function.

```
(set cn
  (lambda (n)
    (lambda (f)
      (lambda (a)
        (if (= n 0)
            a
            (f (((cn (- n 1)) f) a)))))))
```

   What is the result, if any, of the following Scheme expression?

```
(((cn 3) (lambda (x) (* x 4))) 5)
```

2. (10 points) Fill in the blanks in the following Scheme definition of a function `ip`. The function `ip` takes an integer `n` and a list of integers `v` and returns the sum of the product of `n` with each of the elements of `v`. For example `(ip 3 '())` returns 0 and `(ip 4 '(1 2 3))` returns $4 \times 1 + 4 \times 2 + 4 \times 3$ (that is, 24).

```
(set ip
  (lambda (n v)
    ((combine
      ____________________________________
      ____________________________________
      ____________________________________
      )
     ____________________________________
     )))

(set combine (lambda (f sum zero)
     (lambda (l) (if (null? l) zero
       (sum (f (car l)) ((combine f sum zero) (cdr l)))))))
```

3. Consider the following Scheme program:

```
(set s 7)
(set f (lambda (x) (* x s)))
(set g (lambda (s) (f (- s 11))))
```

   (a) (2 points) What is the value of `(g 12)` in a language with static scoping?

   (b) (3 points) What is the value of `(g 12)` in a language with dynamic scoping?

2 General Problem Solving

In this section of the test, you will be directed to solve a problem, and will be required to choose a paradigm and language to solve the problem in, defend your choice, and then solve the problem. The languages you may use are the textbook's: LISP, Scheme, CLU, Smalltalk, and Prolog.

Part of each problem may be to refine the problem statement. Scoring will be one-quarter (1/4) on the choice and defense of paradigm and language (graded as an essay question) and three-quarters (3/4) on the solution itself. Your defense of your choice of paradigm and language should consider the alternatives.

1. (20 points) In this problem you will code some functions on vectors (taken from APL). Let a "vector" be a non-empty finite sequence of integers. In LISP, Scheme, or Prolog a vector would be a list; in CLU and Smalltalk we'll consider vectors to be arrays. Code the following functions on vectors:

   - `+/`, which sums the elements of a vector,
   - `*/`, which gives the product of the elements of a vector,
   - `max/`, which gives the maximum element of a vector,
   - `min/`, which gives the minimum element of a vector,
   - `or/`, which is 1 if any elements of the vector are non-zero, and 0 otherwise.

   If applied to a one-element vector, these all return the value in that element. For example, in Prolog,

   ```
   -> (infer? (or/ (cons 3 nil) X) (print answer is X))
   answer is 1
   ```

   For example, in Smalltalk,

   ```
   -> (set myA (mkArray 1 3))
   -> (at:put: myA 1 20)
   -> (at:put: myA 2 30)
   -> (at:put: myA 3 40)
   -> (+/ myA)
   90
   ```

2. (25 points) In this problem you will program part of a variant of Conway's "game of life". Let a "grid" be a non-empty $n \times n$ matrix of "cells", each of which contains either a one (life) or a zero (no life). You may assume that $n > 0$. Your problem is to decide how to represent the grid in the language of your choice, and to code a function or relation or operation, **`next-grid`** that computes a new grid, using the rules of the game of life given below.

   Suppose you've called the grid `w`. Then we'll write `w[i][j]` for the cell of `w` that is in row `i`, and column `j`. Each cell of the grid `w` has four (4) neighbors, one on each side; that is, `w[i][j]` has neighbors: `w[i+1][j]`, `w[i][j+1]`, `w[i-1][j]`, and `w[i][j-1]`. The contents of each cell `w[i][j]` in the output grid depend on the number of neighbors of `w[i][j]` that are alive (contain 1) in the input grid. The rule is that `w[i][j]` is alive in the output grid (i.e., contains 1) if and only if either:

   - `w[i][j]` is alive in the input grid and 1 or 2 of its neighbors are alive in the input grid, or
   - `w[i][j]` is dead in the input grid and exactly 2 of its neighbors are alive in the input grid.

   To eliminate special cases, you can assume that the grid you get as an argument is a size $(n+2) \times (n+2)$ grid, with the outer cells all permanently zero (dead). This makes all cells have the same neighbors. So your function (or operation or relation) **`next-grid`** has to compute a new $(n+2) \times (n+2)$ grid, by using the rules on the inner $n \times n$ grid.

   Remember to (a) write down your choice of language, and (b) defend that choice against any alternatives. Also explicitly say (c) how you are representing the grids. Be sure to include any code to implement the grids (e.g., to access and update cells) that you use.