Spring, 1992 Name: ________________________________

CS 541 — Programming Languages 1

Final Exam
(Version 1)

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

If you get an incomplete on this test, you can get at most a 'B' on the makeup test.

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 of the following built-in predicates in Prolog:

  `+, -, *, /, is, =:=, <, >, =<, >=, =, true, !, not, fail`

- anything documented in the *Overview and Specification of the built-in types in Little Smalltalk.*

- any of the predefined identifiers in Standard ML, and anything that is coded in *ML for the Working Programmer*, provided you tell use the page number where to find the code in the form: "Using facti from page 56."

1 Denotational Semantics

Consider the following denotational semantics of FL.

Abstract Syntax:
P ∈ Program
E ∈ Expression
I ∈ Identifier
A ∈ ArithOp
R ∈ RelOp
N ∈ Number
B ∈ BooleanLit

P ::= E
E ::= I | N | B | (A E_1 E_2) | (R E_1 E_2)
 | (`if` E_1 E_2 E_3) | (`rec` I E)
 | (`proc` I E) | (`call` E_1 E_2)

Semantic Algebras:
$t \in Tr = Boolean$
$i \in Int = \{0, 1, -1, 2, -2, \ldots\}$
$Error = Unit$
$e \in \textit{Expressible-value} = (Int + Tr + Procedure + Error)_\perp$
$p \in Procedure = \textit{Expressible-value} \to \textit{Expressible-value}$
$Unbound = Unit$
$u \in Environment = \text{Identifier} \to (\textit{Expressible-value} + Unbound)$

Auxiliary Operations
$emptyenv : Environment$
$emptyenv = \lambda I\,.\,\text{in}Unbound()$

$lookup : Environment \to Environment$
$lookup = \lambda u\,.\,\lambda I\,.\,u[\![I]\!]$

$extendenv : Environment \to \text{Identifier} \to \textit{Expressible-value} \to Environment$
$extendenv = \lambda u\,.\,\lambda I\,.\,\lambda e\,.\,\lambda I'\,.\,([\![I]\!] \equiv [\![I']\!]) \to \text{in}\textit{Expressible-value}(e) [\!] u[\![I']\!]$

1. (5 points) Could the definition of *lookup*, $lookup = \lambda u \,.\, \lambda I \,.\, u[\![I]\!]$, be simplified at all? If not, why not? If so, explain your simplification.

Valuation functions:
$\mathcal{P}$: Program $\rightarrow$ *Expressible-value*
$\mathcal{P}[\![\mathrm{E}]\!] = \mathcal{E}[\![\mathrm{E}]\!]\ emptyenv$

$\mathcal{E}$: Expression $\rightarrow$ *Environment* $\rightarrow$ *Expressible-value*

$$
\begin{array}{l}
\mathcal{E}[\![\mathrm{I}]\!] = \lambda u\,.\\
\quad \textbf{cases}\ (lookup\ u\ [\![\mathrm{I}]\!])\ \textbf{of}\\
\quad\quad \mathrm{is}\mathit{Expressible\text{-}value}(e)\ \rightarrow\ e\\
\quad\quad [\!]\mathrm{is}\mathit{Unbound}()\ \rightarrow\ (\mathrm{in}\mathit{Error}())\\
\quad \textbf{end}\\
\mathcal{E}[\![\mathrm{N}]\!] = \lambda u\,.\,\mathrm{in}\mathit{Int}(\mathcal{I}[\![N]\!])\\
\mathcal{E}[\![\mathrm{B}]\!] = \lambda u\,.\,\mathrm{in}\mathit{Tr}(\mathcal{B}[\![B]\!])\\
\mathcal{E}[\![(\texttt{if}\ \mathrm{E}_1\ \mathrm{E}_2\ \mathrm{E}_3)]\!] = \lambda u\,.\\
\quad \textbf{cases}\ \mathcal{E}[\![\mathrm{E}_1]\!]u\ \textbf{of}\\
\quad\quad \mathrm{is}\mathit{Tr}(t)\ \rightarrow\ (t \rightarrow (\mathcal{E}[\![\mathrm{E}_2]\!]u)[\!](\mathcal{E}[\![\mathrm{E}_3]\!]u))\\
\quad\quad \textbf{else}\ \mathrm{in}\mathit{Error}()\\
\quad \textbf{end}\\
\mathcal{E}[\![(\mathrm{A}\ \mathrm{E}_1\ \mathrm{E}_2)]\!] = \lambda u\,.\\
\quad \textbf{cases}\ \mathcal{E}[\![\mathrm{E}_1]\!]u\ \textbf{of}\\
\quad\quad \mathrm{is}\mathit{Int}(i_1)\ \rightarrow\ \textbf{cases}\ \mathcal{E}[\![\mathrm{E}_2]\!]u\ \textbf{of}\\
\quad\quad\quad\quad \mathrm{is}\mathit{Int}(i_2)\ \rightarrow\ \mathrm{in}\mathit{Int}(\mathcal{A}[\![A]\!]\ i_1\ i_2)\\
\quad\quad\quad\quad \textbf{else}\ \mathrm{in}\mathit{Error}()\\
\quad\quad\quad \textbf{end}\\
\quad\quad \textbf{else}\ \mathrm{in}\mathit{Error}()\\
\quad \textbf{end}\\
\mathcal{E}[\![(\mathrm{R}\ \mathrm{E}_1\ \mathrm{E}_2)]\!] = \lambda u\,.\\
\quad \textbf{cases}\ \mathcal{E}[\![\mathrm{E}_1]\!]u\ \textbf{of}\\
\quad\quad \mathrm{is}\mathit{Int}(i_1)\ \rightarrow\ \textbf{cases}\ \mathcal{E}[\![\mathrm{E}_2]\!]u\ \textbf{of}\\
\quad\quad\quad\quad \mathrm{is}\mathit{Int}(i_2)\ \rightarrow\ \mathrm{in}\mathit{Tr}(\mathcal{R}[\![R]\!]\ i_1\ i_2)\\
\quad\quad\quad\quad \textbf{else}\ \mathrm{in}\mathit{Error}()\\
\quad\quad\quad \textbf{end}\\
\quad\quad \textbf{else}\ \mathrm{in}\mathit{Error}()\\
\quad \textbf{end}\\
\mathcal{E}[\![(\texttt{proc}\ \mathrm{I}\ \mathrm{E})]\!] = \lambda u\,.\ \mathrm{in}\mathit{Procedure}(\lambda e\,.\,\mathcal{E}[\![\mathrm{E}]\!](extendenv\ u\ [\![I]\!]\ e))\\
\mathcal{E}[\![(\texttt{call}\ \mathrm{E}_1\ \mathrm{E}_2)]\!] = \lambda u\,.\\
\quad \textbf{cases}\ \mathcal{E}[\![\mathrm{E}_1]\!]\ u\ \textbf{of}\\
\quad\quad \mathrm{is}\mathit{Procedure}(p)\ \rightarrow\ p(\mathcal{E}[\![\mathrm{E}_2\ u]\!])\\
\quad\quad \textbf{else}\ \mathrm{in}\mathit{Error}()\\
\quad \textbf{end}\\
\mathcal{E}[\![(\texttt{rec}\ \mathrm{I}\ \mathrm{E})]\!] = \lambda u\,.\ \mathcal{E}[\![\mathrm{E}]\!](\mathit{fix}(\lambda u'\,.\,(extendenv\ u\ [\![I]\!]\ (\mathcal{E}[\![\mathrm{E}]\!]\ u'))))
\end{array}
$$

2. (8 points) In the semantics above, we have omitted the definition of the valuation functions for $\mathcal{I}$ and $\mathcal{B}$ used to define $\mathcal{E}[\![N]\!]$ and $\mathcal{E}[\![B]\!]$. What should the types of (a) $\mathcal{I}$ and (b) $\mathcal{B}$ be?

3. (15 points) Complete the following calculation showing all the interesting steps used to find its simplest possible normal form. You will have to substitute the definitions of the clauses of $\mathcal{E}$, and the auxiliary operations, such as *extendenv* and *lookup*, in order to simplify the expression. (You'll receive no credit for just giving the answer.) To save copying, you can leave subterms unexpanded until you need them. You can apply β conversion several times in one step. You can also use lemmas if you wish.

$$
\begin{array}{ll}
 & \mathcal{E}[\![\texttt{(proc x x)}]\!] \\
= & \langle \text{by definition of } \mathcal{E} \text{ for procedures} \rangle \\
 & \lambda u \,.\, \mathrm{in}\mathit{Procedure}(\lambda e \,.\, \mathcal{E}[\![\texttt{x}]\!](\mathit{extendenv}\ u\ [\![\texttt{x}]\!]\ e)) \\
= & \langle \text{by definition of } \mathcal{E} \text{ for identifiers} \rangle \\
 & \lambda u \,.\, \mathrm{in}\mathit{Procedure}(\lambda e \,.\ (\lambda u \,.\ \ \textbf{cases}\ (\mathit{lookup}\ u\ [\![\texttt{x}]\!])\ \textbf{of} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathrm{is}\mathit{Expressible\text{-}value}(e) \ \rightarrow\ e \\
 & \qquad\qquad\qquad\qquad\qquad\qquad [\!]\,\mathrm{is}\mathit{Unbound}() \ \rightarrow\ (\mathrm{in}\mathit{Error}()) \\
 & \qquad\qquad\qquad\qquad\qquad \textbf{end}) \\
 & \qquad\qquad\qquad\qquad (\mathit{extendenv}\ u\ [\![\texttt{x}]\!]\ e))
\end{array}
$$

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 either Prolog, Smalltalk, or SML.
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. (10 points) In this problem you will code some functions or relations on vectors (taken from APL). Let a "vector" be a non-empty finite sequence of integers. In SML and Prolog a vector would be a list; in Smalltalk we'll consider vectors to be arrays. (If you use Smalltalk, say what class each method is added to; you may make your code methods of a class `Answer`. Also you can change the names to conform to Smalltalk syntax if needed.) Code the following functions on vectors:

   - `plusRed`, which sums the elements of a vector,
   - `timesRed`, which gives the product of the elements of a vector,
   - `maxRed`, which gives the maximum element of a vector,
   - `minRed`, which gives the minimum element of a vector,
   - `bitOrRed`, which is 1 if at least one element of the vector is non-zero, and zero otherwise.

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

   ```
   ?- bitOrRed([3], X).
   X = 1
   ```

   For example, in Smalltalk,

   ```
   >     #(1 3 5) plusRed
   9
   ```

2. (25 points) In trying to break a secret code, it is convenient to look at how many times a letter occurs in a piece of coded text. We will consider a text to be a list of characters. In this problem you are to write a function or relation or operation `frequencies` that returns a list of 2-element lists (called "pairs"), of letters and how many times that letter occurs, ordered so that the most frequent letter's pair is listed first, the next most frequent letter's pair next, and so on. In SML these pairs should actually be pairs instead of two element lists. Ties are broken by sorting pairs with the same frequency by the character. For example, in SML,

```
frequencies(["g", "o", "o", "d", "e", "g", "g", "a"])
```

returns the list

```
[("g", 3), ("o", 2), ("a", 1), ("d", 1), ("e", 1)]
```

If you choose to implement this in Prolog, your relation `frequencies` should be able to be queried as follows.

```
?- frequencies(['g', 'o', 'o', 'd', 'e', 'g', g', 'a'], X).
X = [['g', 3], ['o', 2], ['a', 1], ['d', 1], ['e', 1]]

?- frequencies([], X).
X = []
```

In Smalltalk you would also use lists as the input and output. (If you use Smalltalk, say what class each method is added to; you may make your code methods of a class `Answer`. Also you can change the names to conform to Smalltalk syntax if needed.)

Remember to (a) write down your choice of paradigm and language, and (b) defend that choice against any alternatives.