March 30, 1990 **Name:** _____

CS 541 — Programming Languages 1

Test

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

Answers to essay questions will be judged on their completeness and clarity. However, please do not feel obligated to fill up all the empty space on the page. Conversely, do not feel limited by the space on a page, since you may continue an answer on the back of a page if you wish.

problem	subject	points	score
1	expressive power	13	
2	lambda calculus	10	
3	domain theory	10	
4	operational semantics	15	
5	denotational semantics	52	
Total		100	

This exam (only) counts for 27.5% of your final grade.

1. (13 points) How should one judge a programming language? What makes a programming language "good"?

2. (10 points) Reduce each of the following lambda expressions to normal form if possible, or state that the expression has no normal form. You do not have to give each of the reduction steps, but you may receive partial credit if you list them and make a mistake.

   (Each of the following is 2 points, except the last, which is 4 points)

   (a) $\lambda x \,.\, \lambda y \,.\, zxy$

   (b) $y\lambda y \,.\, (y\lambda x \,.\, xy)$

   (c) $(\lambda x \,.\, xxx)\lambda y \,.\, yy$

   (d) $\lambda c \,.\, (\lambda k \,.\, (kt))(\lambda x \,.\, (\lambda k \,.\, (kf))(\lambda y \,.\, c(axy)))$

3. Consider the lifted flat domain $D = \{a, b, c\}_\perp$, where $\sqsubseteq_D$ relates $\perp$ to a, to b, and to c, but the proper elements are not related to each other.

   (a) (5 points) How many monotonic functions are there from D to D?

   (b) (5 points) How many continuous functions are there from D to D?

4. (15 points) Give a terminal transition system for a deterministic Turing machine with a one-way tape. You should include a description of the set of configurations (Γ), the set of final configurations (T), the axioms and inference rules that define the transition relation ($\Rightarrow$), and both the input ($\mathcal{I}$) and output ($\mathcal{O}$) functions.

   A Turing machine consists of a finite state control, which can be thought of as a partial function δ that takes a state (from a set of states K) and a character read from the tape and returns a new state, a character to write on the tape, and a direction to move (right or left). The machine starts out with an initial tape (a string of characters from some alphabet Σ), its read head positioned at the leftmost character of the tape, and its finite state control in the state q_0. For each move, the machine reads a character from the tape at the current position, and then depending on the current state, writes a character at the current position, moves left or right one character on the tape, and goes to into a new state (as dictated by δ). If δ is not defined for the current state and read character, or if an attempt is made to move off the left end of the tape, then the machine halts.

5. Design a functional language based on the lambda calculus that can perform computations over booleans and natural numbers along the following lines.

   - Programs consist of a (possibly empty) sequence of nonrecursive function definitions, followed by an expression which may use those functions as well as the operations of the built-in types and other language-defined expressions. There is no "input", other than the expression itself.
   - Instead of λ, function abstraction will be indicated by the keyword `fun`.
   - The language should pass arguments by value.
   - Function abstractions are *not* first-class objects, that is they may only be used in expressions by calling them and may not be passed as arguments.
   - There should be an `if` expression.

   Assume that a set of binary operations the natural numbers, such as `=` and `+`, have already been agreed on. These operations are defined on all arguments of the appropriate domains. You do not have to design these operations.

   (Hint: this problem has several parts. You may want to glance at all of them before starting to give answers, so that you can work from the information given in the parts.)

   (a) (5 points) The following is a start on the abstract syntax. Fill in the rest of the abstract syntax with your own design. That is, you should give the abstract syntax of combinations (i.e., applications), including those involving function abstracts, and the `if` expression (The literals include both boolean and numeric literals; you do not have to give their syntax.)

   $P \in$ Program
   $E \in$ Expression
   $I \in$ Identifier
   $L \in$ Literal
   $O \in$ Operator-for-Built-in

   P ::= `fun` I_1 `(` I_2 `)` `=` E `;` P | E
   E ::= L
   | I
   | `(` O E_1 E_2 `)`

(b) The following domain equations will be used for giving a standard semantics to this language.

$$
\begin{array}{rcl}
n & \in & \mathit{Nat} \\
b & \in & \mathit{Boolean} \\
p & \in & \mathit{Procedure} = \mathit{Denotable\text{-}value} \to \mathit{Expcont} \to \mathit{Expcont} \\
v & \in & \mathit{Value} = \mathit{Nat} + \mathit{Boolean} \\
e & \in & \mathit{Expressible\text{-}value} = \mathit{Value} + \mathit{Procedure} \\
d & \in & \mathit{Denotable\text{-}value} = \mathit{Value} \\
 & \in & \mathit{ANSWER} = (\mathit{Expressible\text{-}value} + \mathit{Error})_{\perp} \\
 & \in & \mathit{Error} = \mathit{Unit} \\
u & \in & \mathit{Environment} = \text{Identifier} \to (\mathit{Denotable\text{-}value} + \mathit{Unbound}) \\
 & \in & \mathit{Unbound} = \mathit{Unit} \\
k & \in & \mathit{Expcont} = \mathit{Expressible\text{-}value} \to \mathit{ANSWER}
\end{array}
$$

i. (5 points) Why is the domain *Expressible-value* different from the domain *Denotable-value*? (That is, what is the reason for the particular choices given above?)

ii. (5 points) The above domain equations are for a standard semantics. What would the domain equation for *Procedure* be if one wished to give a direct semantics (i.e., not using continuations) instead of a standard semantics for this language?

(c) (5 points) Consider the following auxiliary functions.

$\mathit{extend\text{-}env} : \mathit{Environment} \to \text{Identifier} \to \mathit{Denotable\text{-}value} \to \mathit{Environment}$
$\quad \mathit{extend\text{-}env} = \lambda u\,.\,\lambda I\,.\,\lambda d\,.$
$\qquad \lambda I'\,.\,(\mathit{same\text{-}identifier?}\ I\ I') \to (\text{in}\mathit{Denotable\text{-}value}\ d)[\![(u\ I')$
$\mathit{lookup} : \mathit{Environment} \to \text{Identifier} \to (\mathit{Denotable\text{-}value} + \mathit{Unbound})$
$\quad \mathit{lookup} = \lambda u\,.\,\lambda I\,.\,(u\ I)$
$\mathit{top\text{-}level\text{-}cont} : \mathit{Expcont}$
$\quad \mathit{top\text{-}level\text{-}cont} = \lambda e\,.\,(\text{in}\mathit{Expressible\text{-}value}\ e)$

Give lambda expressions that define the following auxiliary functions that work with those given above.

i. $\mathit{null\text{-}env} : \mathit{Environment}$

ii. $\mathit{error\text{-}cont} : \mathit{Expcont}$

(d) (2 points) Consider the following valuation function signatures.

$\mathcal{P}$: Program $\rightarrow$ *ANSWER*
$\mathcal{D}$: Program $\rightarrow$ *Environment* $\rightarrow$ *Expcont* $\rightarrow$ *Expcont*
$\mathcal{E}$: Expression $\rightarrow$ *Environment* $\rightarrow$ *Expcont* $\rightarrow$ *Expcont*
$\mathcal{O}$: Operator-for-Built-in $\rightarrow$ *Nat* $\rightarrow$ *Nat* $\rightarrow$ *Value*

Given that the clause for the denotation of an expression that is a literal is:

$\mathcal{E}[\![L]\!] = \lambda u\ .\ \lambda k\ .\ (k\ (\mathcal{L}[\![L]\!]))$

what is the signature of $\mathcal{L}$?

(e) Consider the following valuation function clauses, some of which are missing or incomplete.

$\mathcal{P}[\![P]\!] = \mathcal{D}[\![P]\!]$ *null-env top-level-cont*

$\mathcal{D}[\![E]\!] = \mathcal{E}[\![E]\!]$

$\mathcal{E}[\![L]\!] = \lambda u\ .\ \lambda k\ .\ (k\ (\mathcal{L}[\![L]\!]))$

$\mathcal{E}[\![I]\!] = \lambda u\ .\ \lambda k\ .$
cases (*lookup* u $[\![I]\!]$) **of**
is*Denotable-value*(d) $\rightarrow$ (k d)
is*Unbound*() $\rightarrow$ *error-cont*
end

$\mathcal{E}[\![(O\ \ E_1\ \ E_2)]\!] = \lambda u\ .\ \lambda k\ .$
$\mathcal{E}[\![E_1]\!]u(\lambda e_1\ .$ **cases** e_1 **of**
is*Value*(v_1) $\rightarrow$ **cases** v_1 **of**
is*Nat*(n_1) $\rightarrow$ $\mathcal{E}[\![E_2]\!]u(\lambda e_2\ .$ **cases** e_2 **of**
is*Value*(v_2) $\rightarrow$ **cases** v_2 **of**
is*Nat*(n_2) $\rightarrow$ (k (in*Value*(($\mathcal{O}[\![O]\!]$)$n_1 n_2$)))
else *error-cont*
end
else *error-cont*
end)
else *error-cont*
end
else *error-cont*
end)

i. (5 points) The clause for the denotation of the `if` expression is missing. What should it be?

ii. (10 points) The clauses for the denotations of combinations (applications), including combinations involving function abstractions, are missing from the above. What should they be?

iii. (5 points) The clause for the denotation of $\mathcal{D}[\![\texttt{fun } I_1 \texttt{ (} I_2 \texttt{) = } E \texttt{ ; } P]\!]$ is missing above. It should have the form

$$\mathcal{D}[\![\texttt{fun } I_1 \texttt{ (} I_2 \texttt{) = } E \texttt{ ; } P]\!] = \lambda u \,.\, \lambda k \,.\, \mathcal{D}[\![P]\!] \ \ldots \ k$$

where the missing stuff (...) should be an environment that extends the environment u with the denotation of the (nonrecursive) function named I_1. This function takes an argument named I_2 and its body is the expression E. Recall that arguments are passed by value in this language. Complete this clause by stating what should replace ....

(f) (10 points) Describe two improvements to the language that would make it more expressive, yet which would not require major changes to the denotational semantics. For each such improvement describe both how the language would be improved and what changes would have to be made to the denotational semantics.