There are five problems on this prelim. You will have 50 minutes in which to work on the problems. You will likely find some problems easier than others; read all problems before beginning to work, and use your time wisely. The prelim is worth 100 points total. Each problem is worth 20 points; the point breakdown for the sub-parts of the problem is printed with the parts of the problem.

This is an open-book examination; you *may* use the textbook and any notes or papers you have brought to class.

Do all written work on the exam itself. If you are running low on space, write on the back of the exam sheets and be sure to write (OVER) on the front side. It is to your advantage to show your work — we will award partial credit for incorrect solutions that are headed in the right direction. If you feel rushed, try to write a brief statement that captures key ideas relevant to the solution of the problem.

Name ______________________________

| Problem | Points |
|---|---|
| 1 | 20 |
| 2 | 20 |
| 3 | 20 |
| 4 | 20 |
| 5 | 20 |

## 1 Lambda calculus

In class, we studied *Church numerals* as one way of representing natural numbers in the untyped lambda calculus. In this problem, we explore an alternate representation.

Recall the following definitions from the untyped lambda calculus:

$$\begin{aligned} \mathit{TRUE} &\stackrel{def}{=} (\lambda \mathsf{a}\ (\lambda \mathsf{b}\ \mathsf{a})) \\ \mathit{FALSE} &\stackrel{def}{=} (\lambda \mathsf{a}\ (\lambda \mathsf{b}\ \mathsf{b})) \end{aligned}$$

We can define the numeral $\boxed{n}$ inductively as follows:

$$\begin{aligned} \boxed{0} &\stackrel{def}{=} (\lambda \mathsf{z}\ \mathsf{z}) \\ \boxed{n+1} &\stackrel{def}{=} (\lambda \mathsf{s}\ ((\mathsf{s}\ \mathit{FALSE})\ \boxed{n}\ )) \end{aligned}$$

a. (10 pts) Consider the expression $D$ defined as $D \stackrel{def}{=} (\lambda \mathsf{x}\ (\mathsf{x}\ \mathit{FALSE}))$.

Reduce the following expressions to normal form using beta and eta reductions as necessary. You do *not* have to show all intermediate reductions. You may use a term of the form $\boxed{n}$ for your answer to the second expression.

$(D\ \boxed{0}\ )$

$(D\ \boxed{n+1}\ )$

b. (10 pts) Give a definition for a lambda expression *ZERO?* such that

$$\begin{aligned} (\mathit{ZERO?}\ \boxed{0}\ ) &\ \to^* \ \mathit{TRUE} \\ (\mathit{ZERO?}\ \boxed{n+1}\ ) &\ \to^* \ \mathit{FALSE} \end{aligned}$$

You do not have to show that these relations hold: just give the definition of *ZERO?*.

## 2 Parameter passing

Consider the following program written in the $F_0$ language, with let desugared in the usual way.

```
(let (f (fn y 19))
 (let (a 3)
  (let (g (fn b (if (> a 0)
                   (f (+ b #t))
                   (f ((fn x (x x)) (fn x (x x)))))))
   (let (f (fn z 100))
    (let (a -2)
     (g 47))))))
```

What is the value of this program in

a. (5 pts) Call-by-name parameter passing and static scoping

b. (5 pts) Call-by-value parameter passing and static scoping

c. (5 pts) Call-by-name parameter passing and dynamic scoping

d. (5 pts) Call-by-value parameter passing and dynamic scoping

## 3 Domain theory

a. (10 pts)

Noam Wright thinks that the sum domain construction $D_1 + D_2$ is unnecessarily complex. He proposes the following way of defining a sum domain without using injection functions. Let $\sqsubseteq_1$ and $\sqsubseteq_2$ be the ordering relations for the domains $D_1$ and $D_2$. Noam suggests that the sum of the domains $(D_1, \sqsubseteq_1)$ and $(D_2, \sqsubseteq_2)$ could be defined in general as $(D_1 \cup D_2, \{(d_1, d_2) \mid (d_1, d_2) \in D_1 \vee (d_1, d_2) \in D_2\})$. What's wrong with this construction?

b. (10 pts)

In the denotational semantics for $F_!$, we relied on a function *alloc* that for any state $\sigma$, returns a location $l$ such that $\sigma(l) = \mathit{Unbound}$. Let us also suppose that if the store is full, *alloc* diverges: it maps an element of *Store* to an element of $\mathit{Loc}_\perp$, where $\mathit{Store} = \mathit{Loc} \to (\mathit{Value} + \mathit{Unbound})$. However, the function *alloc* is not one of the several meta-language constructs that have been shown to lead to sound language definitions. What properties does *alloc* need to permit its use? Show that *any* function *alloc* meeting the specification above has these properties.

## 4 Operational semantics and state

a. (10 pts) Suppose we extend the $F_!$ language with a simple transaction mechanism. The new kind of expression (transaction $e$) evaluates the expression $e$ using the current store. If $e$ evaluates to #t, the whole expression has the same result and effect on the state as $e$. If $e$ evaluates to #f, the transaction expression has the result #f, but has *no effect on the state.*

For example, the expression (transaction (seq (a := 1) #t)) is equivalent to (seq (a := 1) #t), but the expression (transaction (seq (a := 1) #f)) is equivalent to #f; any modification to the store location a is erased by the transaction expression.

Write the necessary large-step rules for evaluating this new kind of expression. You may ignore the case in which $e$ evaluates to a value other than #t or #f.

b. (10 pts) Now Noam Wright wishes to write small-step semantics for the transaction expression. He proposes that the semantics be written using the usual configurations $\langle e, \sigma \rangle$, and a new kind of configuration $\langle e, \sigma, \sigma' \rangle$ be added to support transactions. The idea is that in this configuration, $\sigma'$ stores the state that will be restored if the current transaction fails. He then writes the following semantics for transaction:

$$\frac{}{\langle \textsf{transaction } e, \sigma \rangle \to_1 \langle e, \sigma, \sigma \rangle} \qquad \frac{\langle e, \sigma \rangle \to_1 \langle e', \sigma'' \rangle}{\langle e, \sigma, \sigma' \rangle \to_1 \langle e', \sigma'', \sigma' \rangle}$$

$$\frac{}{\langle \#\textsf{t}, \sigma, \sigma' \rangle \to_1 \langle \#\textsf{t}, \sigma \rangle} \qquad \frac{}{\langle \#\textsf{f}, \sigma, \sigma' \rangle \to_1 \langle \#\textsf{f}, \sigma' \rangle}$$

Explain what is wrong with this semantics and provide an example of an expression that is evaluated improperly by it.

## 5 Denotational semantics and continuations

In this problem, we will explore a denotational semantics for a simple coroutine mechanism that extends $F_!$ (or $F_0$). This will involve adding two new expressions to the language. First, the new expression form (cobegin $e_1$ $e_2$) starts a pair of threads, or coroutines, which evaluate $e_1$ and $e_2$ in parallel respectively. When one of these expressions completes evaluation, the whole cobegin expression completes with the value of that expression. The thread evaluating the first expression $e_1$ is always the first one that runs. Each of the coroutines runs continuously until it evaluates a (yield) expression, which transfers control to the other coroutine. The result of this second new expression (yield) is always *unit*, although the result does not arrive until the *other* coroutine evaluates (yield) itself. This mechanism can be used to try two different ways of computing some result in parallel, and to select the result that arrives first. (In a more typical coroutine mechanism, more interesting values could be transferred back and forth between the coroutines as they yield control to each other).

Here are some examples of programs using this mechanism:

- (cobegin 1 2) evaluates to 1
- (cobegin (seq 1 (yield) 3) 2) evaluates to 2
- (cobegin (seq 1 (yield) 3) (seq 2 (yield) 4) evaluates to 3.
- The program

```
(let (x (ref 4))  (cobegin (seq (x := (+ 1 (^ x))) (yield) (^ x))
                           (seq (x := (* (^ x) (^ x))) (yield)))
  ))
```

  evaluates to 25.

We can write denotational semantics for this extended language by modifying the signature of the function $\mathcal{C}$ to add an additional argument from the domain *Cont*:

$$\mathcal{C}[\![e]\!] : \mathit{Cont} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Store} \to \mathit{Answer}$$

The first *Cont* argument is the normal continuation; the second *Cont* argument is the continuation that is invoked to transfer control to the other thread running in parallel with this one (if there is any). In addition, we modify the domain *Cont* to allow the other coroutine's continuation to be transferred dynamically so it can be restarted when the next (yield) is encountered in this coroutine:

$$\mathit{Cont} = \mathit{Value} \to \mathit{Cont} \to \mathit{Store} \to \mathit{Answer}$$

With these definitions, most of the semantics of $F_0$ or $F_!$ can be written in the same fashion as before, thanks to the magic of eta reductions. For example, consider integer constants and the seq expression:

$$\begin{aligned}
\mathcal{C}[\![n]\!]k\rho &= (\lambda k_c\sigma \,.\, knk_c\sigma) = kn \\
\mathcal{C}[\![\mathsf{seq}\ e_1\ e_2]\!]k\rho &= \mathcal{C}[\![e_1]\!](\lambda v \,.\, \mathcal{C}[\![e_2]\!]k\rho)\rho
\end{aligned}$$

Of course, semantics are needed for the new kinds of expressions. We define the denotational semantics for cobegin and yield as follows, using eta reductions to avoid talking about the store.

$$\begin{aligned}
\mathcal{C}[\![\mathsf{cobegin}\ e_1\ e_2]\!]k\rho k_c &= \mathcal{C}[\![e_1]\!]k\rho(\lambda u k'_c \,.\, \mathcal{C}[\![e_2]\!]k\rho k'_c) \\
\mathcal{C}[\![\mathsf{yield}]\!]k\rho k_c &= k_c(\mathit{unit})k
\end{aligned}$$

We also need to define the meaning of an entire program $e$. This will be defined as $\mathcal{C}[\![e]\!]k_0\rho_0k_e\sigma_0$, where $k_0 = \lambda vk_c\sigma.\langle v, \sigma\rangle$ and $k_e = \lambda vk_c\sigma.error$.

Using these semantics, what is the result of evaluating the following programs?

a. (3 pts) (yield)

b. (3 pts) (cobegin (yield) (yield))

c. (3 pts) (cobegin (seq (yield) 1) (cobegin (seq (yield) 2) (seq (yield) 3)))

The rules for function definition and application, and the domain for function values must be changed.

d. (5 pts) What is an appropriate re-definition of the domain *Function*?

e. (6 pts) Complete this definition accordingly:

$\mathcal{C}[\![\text{fn x e}]\!]k\rho =$