MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

1998 Midterm and Solutions

There are *four* problems on this examination. They are followed by an appendix that contains reference material from the course notes. *The appendix contains no problems;* it is just a handy reference.

You will have eighty-five minutes in which to work the problems. Some problems are easier than others: read all problems before beginning to work, and use your time wisely!

This examination is open-book: you may use whatever reference books or papers you have brought to class. The number of points awarded for each problem is placed in brackets next to the problem number. There are 100 points total on the exam.

Do all written work in your examination booklet – we will not collect the examination handout itself; you will only be graded for what appears in your examination booklet. It will be to your advantage to show your work – we will award partial credit for incorrect solutions that make use of the right techniques.

If you feel rushed, be sure to write a brief statement indicating the key idea you expect to use in your solutions. We understand time pressure, but we can't read your mind.

This examination has text printed on only one side of each page. Rather than flipping back and forth between pages, you may find it helpful to rip pages out of the exam so that you can look at more than one page at the same time.

Appendix:

The figures in the Appendix are very similar to the ones in the course notes. Some bugs have been fixed, and some figures have been simplified to remove parts inessential for this exam. You will not be marked down if you use the corresponding figures in the course notes instead of the appendices.

Problem 1: The Two Game [10 points]

Indicate whether each of the following expressions in the specified FL language-variant evaluates to the number two (2).

a. [2 points] Dynamically-scoped Call-By-Name FL

```
(let ((foo (let ((a 5))
             (lambda (x) (- a x)))))
   (let ((b 3))
      (foo b)))
```

b. [2 points] Statically-scoped Call-By-Reference FLAVAR!

```
(let ((x 0)
      (foo (lambda (y z)
             (begin (set! y 2)
                    (/ 4 z)))))
   (foo x x))
```

c. [2 points] Statically-scoped Call-By-Value FL

```
(let ((foo (proc x (rec x x)))
      (bar (proc y 2)))
   (bar foo))
```

d. [2 points] Statically-scoped Call-By-Value FLK! (with `label` and `jump`)

```
(label bar
   (* 2 (label foo
          (jump foo (jump bar 1)))))
```

e. [2 points] Dynamically-scoped Call-By-Value FL

```
(let ((x 1))
   (let ((f (lambda () x))
         (g (lambda (h) (h))))
      (let ((x 2))
         (g f))))
```

Problem 2: Operational Semantics [30 points]

YOUR ANSWERS TO THIS QUESTION SHOULD BE BASED ON THE FLK SOS AS PRESENTED IN APPENDIX A.

Louis Reasoner has an idea for a new FLK command, `(terminate` E_1 E_2`)`. If either E_1 or E_2 terminates with a value or an error, `(terminate` E_1 E_2`)` will also terminate with a value or an error. In evaluating `terminate`, we run one expression for one step, then run the other expression for one step, and so on. Louis initially worked out a few examples of how his new construct would work:

`(terminate 1 2)` $\Rightarrow$ *1* **or** *2* **(implementation dependent)**
`(terminate 2 (rec x x))` $\Rightarrow$ *2*
`(terminate 1 (call 3 0))` $\Rightarrow$ *1* **or** *error: can't apply non-procedure* **(implementation dependent)**
`(terminate (rec x x) (/ 3 0))` $\Rightarrow$ *error: divide by zero*
`(terminate (rec x x) (rec x x))` $\Rightarrow \perp$

Louis is really excited about the `terminate` construct. His old implementation of FLK required him to reboot any time his program ran into an infinite loop. Although he hasn't solved the halting problem, now he can guarantee not to have to reboot (excepting, of course, when his new-fangled operating system crashes) by testing his programs with terminate and his new `(timer` N`)` construct.

Louis defined the following transition rule(s) for `timer`:

$$\begin{array}{rl} \texttt{(timer}\ N_1\texttt{)} \Rightarrow \texttt{(timer}\ N_2\texttt{)} & \\ \quad \text{where}\ \ N_1 > 1 & \text{[}\textit{timer-countdown}\text{]} \\ \quad \text{and}\ \ N_2 = N_1 - 1 & \end{array}$$

$$\texttt{(timer 1)} \Rightarrow \#u \qquad \text{[}\textit{timer}\text{]}$$

Louis can now use the `terminate` construct to run his program *might-go-infinite* for exactly 1000 steps (where we consider each transition to be one step). The following expression will return the result of *might-go-infinite* if it completes in under 1000 steps, otherwise it returns `#u`.

```
(terminate (timer 1000) might-go-infinite)
```

Unfortunately, Louis set off for Hawaii before he was able to extend the FL Operational Semantics to include `terminate`. In his absence, you are asked to finish it up.

a. [15 points] Give the transition rules for `terminate`.

b. [5 points] Are your rules confluent?

c. [10 points] Show how the following expression would be evaluated using the rules above:

```
(terminate (call (proc x (primop + x 2)) 5)
           (if (> 3 4)
               (rec x x)
               (proc y 1)))
```

Problem 3: Denotational Semantics and Mutation [30 points]

YOUR ANSWERS TO THIS PROBLEM SHOULD BE BASED ON THE STANDARD DENOTATIONAL SEMANTICS FOR FLK! AS PRESENTED IN APPENDIX B.

Alyssa P. Hacker has decided to add a new feature to FLK!: the ability to undo store modifications. She introduces a single new form into FLK! called `undo!`:

```
(undo!)   ; undo the last mutation to the store
```

Here is an example of how `undo!` can be used to undo a `cell-set!` using FL! sugar:

```
(let ((c (cell 0)))
   (begin
      (primop cell-set! c 1)
      (undo!)
      (primop cell-ref c)))
```

$\Rightarrow$ *0*

Before Alyssa is called away to testify at a large antitrust trial, she managed to update the Standard Semantics found in Appendix B as follows:

$$\begin{aligned} \text{Cmdcont} &= \text{StoreStack} \to \text{Answer} \\ z \in \text{StoreStack} &= \text{Store}^* \end{aligned}$$

$$\mathcal{TL}[\![E]\!] = \mathcal{E}[\![E]\!]\ \textit{empty-env}\ \textit{top-level-cont}\ \textit{empty-store}.[]_{Store}$$

You can assume all of the semantic algebra helper functions that deal in CmdCont have been properly updated. Store operations *same-location?*, *next-location*, *empty-store*, *fetch*, *assign*, *fresh-loc*, and *first-fresh* remain unchanged.

a. [10 points] As she was running out the door, Alyssa mumbled something about the dangers of undoing cell creations using `undo!`. Give an example of a program that would generate a run-time error if Alyssa permits the `cell` form to be undone.

b. [10 points] Dissatisfied with the possibility of run-time errors, Alyssa faxes you a revised meaning function for `cell` and assures you that we can prevent undos to cell creations.

$$\begin{aligned} \mathcal{E}[\![\texttt{(primop cell E)}]\!] = \lambda ek\ .\ \mathcal{E}[\![\texttt{E}]\!]\ \text{e}\ (\lambda vz\ .\ & \textbf{matching}\ z \\ & \triangleright s.z_1\ [\!]\ (k\ (\text{Location} \mapsto \text{Value}\ \ (\textit{fresh-loc}\ \ s)) \\ & \qquad (\textit{assign-to-all}\ \ (\textit{fresh-loc}\ \ s)\ \ v\ \ z)) \\ & \textbf{endmatching}\ ) \end{aligned}$$

$$\begin{aligned} & \textit{assign-to-all} : \text{Location} \to \text{Storable} \to \text{StoreStack} \to \text{StoreStack} \\ & = \textbf{fix}_{\text{Location} \to \text{Storable} \to \text{StoreStack} \to \text{StoreStack}}\ \lambda f\ .\ (\lambda l \sigma z\ .\ \textbf{matching}\ z \\ & \qquad \triangleright []_{Store}\ [\!]\ []_{Store} \\ & \qquad \triangleright s.z_1\ [\!]\ (\textit{assign}\ \ l\ \ \sigma\ \ s).\ (f\ l\ \sigma\ z_1) \\ & \qquad \textbf{endmatching}\ ) \end{aligned}$$

Provide a revised meaning function for $\mathcal{E}[\![\texttt{(primop cell-set!}\ \ E_1\ \ E_2\texttt{)}]\!]$.

c. [10 points] Also provide a revised meaning function for Alyssa's new `undo!` form, $\mathcal{E}[\![\texttt{undo!)}]\!]$. Her notes specified that even in the absense of `cell-set!`'s, `undo!` should not produce any errors.

Problem 4: Denotational Semantics: Control [30 points]

YOUR ANSWERS TO THIS PROBLEM SHOULD BE BASED ON THE STANDARD DENOTATIONAL SEMANTICS FOR FLK! AS PRESENTED IN APPENDIX B.

Ben Bitdiddle is now ensconced in a major research university where he's been fooling around with loops in FLK!. Ben has decided to add the following features to FLK!:

$$E ::= \ldots \text{all FLK! forms} \ldots \mid \texttt{(loop } E\texttt{)} \mid \texttt{(exit } E\texttt{)} \mid \texttt{(jump)}$$

Here's an example of how Ben's new loop construct would work:

```
(let ((c (cell 0)))
   (loop
      (begin
         (primop cell-set! c (+ (primop cell-ref c) 1))
         (if (> (cell-ref c) 10)
            (exit (primop cell-ref c))
            (jump)))))
```

$\Rightarrow$ `11`

As one of Ben's grad students, your job is to write the denotational semantics for FLK! with his new looping functionality. Ben has already revised the domain equations for you:

$$\begin{aligned} j \in \text{Jumpcont} &= \text{Cmdcont} \\ x \in \quad \text{Exitcont} &= \text{Expcont} \end{aligned}$$

He's also changed the signature of the $\mathcal{E}$ meaning function so that every expression is evaluated with both a jump and an exit continuation:

$$\mathcal{E} \; : \text{Exp} \rightarrow \text{Environment} \rightarrow \text{Jumpcont} \rightarrow \text{Exitcont} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont}$$

Ben did give you the meaning function clause for $\mathcal{E}[\![\texttt{(jump)}]\!]$ for reference.

$$\mathcal{E}[\![\texttt{(jump)}]\!] = \lambda ejxk \,.\, j$$

a. [10 points] Extend the meaning function in the FLK! Standard Semantics as given in Appendix B to include $\mathcal{E}[\![\texttt{(loop } E\texttt{)}]\!]$, given the new domains.

b. [10 points] Define the meaning function clause for $\mathcal{E}[\![\texttt{(exit } E\texttt{)}]\!]$.

c. [10 points] Show that `(loop (jump))` is equivalent to bottom.

A An Operational Semantics for FLK

E	$\in$	Exp		
L	$\in$	Lit		
I	$\in$	Identifier		
B	$\in$	Boollit	=	$\{\texttt{\#t}, \texttt{\#f}\}$
N	$\in$	Intlit	=	$\{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
O	$\in$	Primop	=	;; *Defined by standard library.*
V	$\in$	Value	=	$\{\texttt{\#u}\} \cup$ Boollit $\cup$ Intlit
				$\cup \{\texttt{(symbol } I\texttt{)}\} \cup \{\texttt{(proc } I\ E\texttt{)}\} \cup \{\texttt{(pair } E_1\ E_2\texttt{)}\}$

E	::=	L	[Literal]
	|	I	[Variable Reference]
	|	$\texttt{(primop } O_{name}\ E_{arg}\texttt{*)}$	[Primitive Application]
	|	$\texttt{(proc } I_{formal}\ E_{body}\texttt{)}$	[Abstraction]
	|	$\texttt{(call } E_{operator}\ E_{operand}\texttt{)}$	[Application]
	|	$\texttt{(if } E_{test}\ E_{consequent}\ E_{alternate}\texttt{)}$	[Branch]
	|	$\texttt{(pair } E_{left}\ E_{right}\texttt{)}$	[Pairing]
	|	$\texttt{(rec } I_{var}\ E_{body}\texttt{)}$	[Recursion]
L	::=	`#u`	[Unit Literal]
	|	B	[Boolean Literal]
	|	N	[Integer Literal]
	|	$\texttt{(symbol } I\texttt{)}$	[Symbolic Literal]

Figure 1: An s-expression grammar for FLK

$$\texttt{(call (proc } I\ E_1\texttt{)}\ E_2\texttt{)} \Rightarrow [E_2/I]E_1 \qquad [\textit{call-apply}]$$

$$\frac{E_1 \Rightarrow E_1'}{\texttt{(call } E_1\ E_2\texttt{)} \Rightarrow \texttt{(call } E_1'\ E_2\texttt{)}} \qquad [\textit{call-operator}]$$

$$\texttt{(if \#t } E_1\ E_2\texttt{)} \Rightarrow E_1 \qquad [\textit{if-true}]$$

$$\texttt{(if \#f } E_1\ E_2\texttt{)} \Rightarrow E_2 \qquad [\textit{if-false}]$$

$$\frac{E_1 \Rightarrow E_1'}{\texttt{(if } E_1\ E_2\ E_3\texttt{)} \Rightarrow \texttt{(if } E_1'\ E_2\ E_3\texttt{)}} \qquad [\textit{if-test}]$$

$$\texttt{(rec } I\ E\texttt{)} \Rightarrow [\texttt{(rec } I\ E\texttt{)}/I]E \qquad [\textit{rec}]$$

$$\frac{E \Rightarrow E'}{\texttt{(primop } O\ E\texttt{)} \Rightarrow \texttt{(primop } O\ E'\texttt{)}} \qquad [\textit{unary-arg}]$$

$$\frac{E_1 \Rightarrow E_1'}{\texttt{(primop } O\ E_1\ E_2\texttt{)} \Rightarrow \texttt{(primop } O\ E_1'\ E_2\texttt{)}} \qquad [\textit{binary-arg-1}]$$

$$\frac{E_2 \Rightarrow E_2'}{\texttt{(primop } O\ V\ E_2\texttt{)} \Rightarrow \texttt{(primop } O\ V\ E_2'\texttt{)}} \qquad [\textit{binary-arg-2}]$$

Figure 2: A partial set of FLK rewrite rules

B Standard Semantics of FLK!

$v \in \text{Value} = \text{Unit} + \text{Bool} + \text{Int} + \text{Sym} + \text{Pair} + \text{Procedure} + \text{Location}$
$k \in \text{Expcont} = \text{Value} \to \text{Cmdcont}$
$\gamma \in \text{Cmdcont} = \text{Store} \to \text{Answer}$
$\text{Answer} = (\text{Value} + \text{Error})_\perp$
$\text{Error} = \text{Sym}$
$p \in \text{Procedure} = \text{Denotable} \to \text{Expcont} \to \text{Cmdcont}$
$d \in \text{Denotable} = \text{Value}$
$e \in \text{Environment} = \text{Identifier} \to \text{Binding}$
$\beta \in \text{Binding} = (\text{Denotable} + \text{Unbound})_\perp$
$\text{Unbound} = \{\textit{unbound}\}$
$s \in \text{Store} = \text{Location} \to \text{Assignment}$
$l \in \text{Location} = \text{Nat}$
$\text{Assignment} = (\text{Storable} + \text{Unassigned})_\perp$
$\sigma \in \text{Storable} = \text{Value}$
$\text{Unassigned} = \{\textit{unassigned}\}$

top-level-cont : Expcont
$= \lambda v\,.\ \lambda s\,.\ (\text{Value} \mapsto \text{Answer}\ \ v)$
error-cont : Error → Cmdcont
$= \lambda y\,.\ \lambda s\,.\ (\text{Error} \mapsto \text{Answer}\ \ y)$

empty-env : Environment $= \lambda I\,.\ (\text{Unbound} \mapsto \text{Binding}\ \ \textit{unbound})$

test-boolean : (Bool → Cmdcont) → Expcont
$= \lambda f\,.\ (\lambda v\,.$ **matching** v
$\triangleright\ (\text{Bool} \mapsto \text{Value}\ \ b) \llbracket\ (f\ \ b)$
$\triangleright$ **else** (*error-cont* `non-boolean`)
endmatching)
Similarly for:
test-procedure : (Procedure → Cmdcont) → Expcont
test-location : (Location → Cmdcont) → Expcont
etc.

ensure-bound : Binding → Expcont → Cmdcont
$= \lambda \beta k\,.$ **matching** β
$\triangleright\ (\text{Denotable} \mapsto \text{Binding}\ \ v) \llbracket\ (k\ \ v)$
$\triangleright\ (\text{Unbound} \mapsto \text{Binding}\ \ \textit{unbound}) \llbracket\ (\textit{error-cont}$ `unbound-variable`)
endmatching
Similarly for:
ensure-assigned : Assignment → Expcont → Cmdcont

Figure 3: Semantic algebras for standard semantics of strict CBV FLK!.

```
same-location? : Location → Location → Bool = λl1 l2 . (l1 =Nat l2)
next-location : Location → Location = λl . (l +Nat 1)

empty-store : Store = λl . (Unassigned ↦Assignment unassigned)

fetch : Location → Store → Assignment = λls . (s l)

assign : Location → Storable → Store → Store
= λl1 σ s . λl2 . if (same-location? l1 l2)
                 then (Storable ↦Assignment σ)
                 else (fetch l2 s)

fresh-loc : Store → Location = λs . (first-fresh s 0)

first-fresh : Store → Location → Location
= λs l . matching (fetch l s)
         ▷ (Unassigned ↦Assignment unassigned) ⟦ l
         ▷ else (first-fresh s (next-location l))
         endmatching
```

Figure 4: Store helper functions for standard semantics of strict CBV FLK!.

$\mathcal{TL}$: Exp → Answer
$\mathcal{E}$: Exp → Environment → Expcont → Cmdcont
$\mathcal{L}$: Lit → Value *; Defined as usual*

$\mathcal{TL}[\![E]\!] = \mathcal{E}[\![E]\!]$ *empty-env top-level-cont empty-store*

$\mathcal{E}[\![L]\!] = \lambda ek\,.\ k\ \mathcal{L}[\![L]\!]$

$\mathcal{E}[\![I]\!] = \lambda ek\,.$ *ensure-bound* (*lookup* $e\ I$) k

$\mathcal{E}[\![$`(proc` $I\ E$`)`$]\!] = \lambda ek\,.\ k$ (Procedure $\mapsto$Value $(\lambda dk'\,.\,\mathcal{E}[\![E]\!]\ [I : d]e\ k')$)

$\mathcal{E}[\![$`(call` $E_1\ E_2$`)`$]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e$ (*test-procedure* $(\lambda p\,.\,\mathcal{E}[\![E_2]\!]\ e\ (\lambda v\,.\,p\ v\ k))$)

$\mathcal{E}[\![$`(if` $E_1\ E_2\ E_3$`)`$]\!] =$
$\lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e$ (*test-boolean* $(\lambda b\,.$ **if** b **then** $\mathcal{E}[\![E_2]\!]\ e\ k$ **else** $\mathcal{E}[\![E_3]\!]\ e\ k)$)

$\mathcal{E}[\![$`(pair` $E_1\ E_2$`)`$]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_1\,.\,\mathcal{E}[\![E_2]\!]\ e\ (\lambda v_2\,.\,k$ (Pair $\mapsto$Value $\langle v_1,\ v_2\rangle))))$

$\mathcal{E}[\![$`(cell` E`)`$]\!] = \lambda ek\,.\ \mathcal{E}[\![E]\!]\ e\ (\lambda vs\,.\ k$ (Location $\mapsto$Value (*fresh-loc* s))
(*assign* (*fresh-loc* s) $v\ s$))

$\mathcal{E}[\![$`(begin` $E_1\ E_2$`)`$]\!] = \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_{\text{ignore}}\,.\,\mathcal{E}[\![E_2]\!]\ e\ k)$

$\mathcal{E}[\![$`(primop cell-ref` E`)`$]\!] = \lambda ek\,.\ \mathcal{E}[\![E]\!]\ e$ (*test-location* $(\lambda ls\,.$ *ensure-assigned* (*fetch* $l\ s$) $k\ s)$)

$\mathcal{E}[\![$`(primop cell-set!` $E_1\ E_2$`)`$]\!]$
$= \lambda ek\,.\ \mathcal{E}[\![E_1]\!]\ e$ (*test-location* $(\lambda l\,.\,\mathcal{E}[\![E_2]\!]\ e\ (\lambda vs\,.\,k$ (Unit $\mapsto$Value *unit*) (*assign* $l\ v\ s$))))

$\mathcal{E}[\![$`(rec` $I\ E$`)`$]\!] = \lambda eks\,.$ let f = $\mathbf{fix}_{\text{Answer}}\ (\lambda a\,.\ \mathcal{E}[\![E]\!]$ [I : (*extract-value* a)] e top-level-cont s)
matching f
▷ (Value $\mapsto$Answer v) ‖ $\mathcal{E}[\![E]\!]$ [$I : v$] e $k\ s$
▷ **else** f
endmatching

extract-value : Answer → Binding
$= \lambda a\,.$ **matching** a
▷ (Value $\mapsto$Expressible v) ‖ (Denotable $\mapsto$Binding v)
▷ **else** $\perp_{\text{Binding}}$
endmatching

Figure 5: Valuation clauses for standard semantics of strict CBV FLK!.

Midterm Solutions

Problem 1: The Two Game [10 points]

Indicate whether each of the following expressions in the specified FL language-variant evaluates to the number two (2).

a. [2 points] Dynamically-scoped Call-By-Name FL

```
(let ((foo (let ((a 5))
             (lambda (x) (- a x)))))
   (let ((b 3))
      (foo b)))
```

Solution: No, the result is *error: a is undefined*.

b. [2 points] Statically-scoped Call-By-Reference FLAVAR!

```
(let ((x 0)
      (foo (lambda (y z)
             (begin (set! y 2)
                    (/ 4 z)))))
   (foo x x))
```

Solution: Yes, aliasing allows us to use multiple names for the same location in the store.

c. [2 points] Statically-scoped Call-By-Value FL

```
(let ((foo (proc x (rec x x)))
      (bar (proc y 2)))
   (bar foo))
```

Solution: Yes, even in Call-By-Value we don't evaluate the body of a procedure until it's called.

d. [2 points] Statically-scoped Call-By-Value FLK! (with `label` and `jump`)

```
(label bar
   (* 2 (label foo
          (jump foo (jump bar 1)))))
```

Solution: No, `(jump bar 1)` passes `1` directly to the label bar, so the result is `1`.

e. [2 points] Dynamically-scoped Call-By-Value FL

```
(let ((x 1))
   (let ((f (lambda () x))
         (g (lambda (h) (h))))
      (let ((x 2))
         (g f))))
```

Solution: Yes, procedure f gets the value 2 for the free variable x from the dynamic environment.

Problem 2: Operational Semantics [30 points]

Louis Reasoner has an idea for a new FLK command, `(terminate` E_1 E_2`)`. If either E_1 or E_2 terminates with a value or an error, `(terminate` E_1 E_2`)` will also terminate with a value or an error. In evaluating `terminate`, we run one expression for one step, then run the other expression for one step, and so on. Louis initially worked out a few examples of how his new construct would work:

`(terminate 1 2)` $\Rightarrow$ *1* **or** *2* **(implementation dependent)**
`(terminate 2 (rec x x))` $\Rightarrow$ *2*
`(terminate 1 (call 3 0))` $\Rightarrow$ *1* **or** *error: can't apply non-procedure* **(implementation dependent)**
`(terminate (rec x x) (/ 3 0))` $\Rightarrow$ *error: divide by zero*
`(terminate (rec x x) (rec x x))` $\Rightarrow \bot$

Louis is really excited about the `terminate` construct. His old implementation of FLK required him to reboot any time his program ran into an infinite loop. Although he hasn't solved the halting problem, now he can guarantee not to have to reboot (excepting, of course, when his new-fangled operating system crashes) by testing his programs with terminate and his new `(timer` N`)` construct.

Louis defined the following transition rule(s) for `timer`:

$$\texttt{(timer}\ N_1\texttt{)} \Rightarrow \texttt{(timer}\ N_2\texttt{)} \quad \text{where } N_1 > 1 \text{ and } N_2 = N_1 - 1 \qquad \text{[\textit{timer-countdown}]}$$

$$\texttt{(timer 1)} \Rightarrow \#u \qquad \text{[\textit{timer}]}$$

Louis can now use the `terminate` construct to run his program *might-go-infinite* for exactly 1000 steps (where we consider each transition to be one step). The following expression will return the result of *might-go-infinite* if it completes in under 1000 steps, otherwise it returns `#u`.

```
(terminate (timer 1000) might-go-infinite)
```

Unfortunately, Louis set off for Hawaii before he was able to extend the FL Operational Semantics to include `terminate`. In his absence, you are asked to finish it up.

a. [15 points] Give the transition rules for `terminate`.

Solution:

$$\frac{E_1 \Rightarrow E_1'}{\texttt{(terminate}\ E_1\ E_2\texttt{)} \Rightarrow \texttt{(terminate}\ E_2\ E_1'\texttt{)}} \qquad \text{[\textit{terminate-progress}]}$$

$$\texttt{(terminate}\ V\ E\texttt{)} \Rightarrow V \qquad \text{[\textit{terminate-with-value}]}$$

b. [5 points] Are your rules confluent?

Solution: Yes, the rules are confluent.

c. [10 points] Show how the following expression would be evaluated using the rules above:

```
(terminate (call (proc x (primop + x 2)) 5)
           (if (> 3 4)
               (rec x x)
               (proc y 1)))
```

Solution:

```
⇒ (terminate (if (> 3 4)
                 (rec x x)
                 (proc y 1))
             (primop + 5 2))

⇒ (terminate (primop + 5 2)
             (if false
                 (rec x x)
                 (proc y 1)))

⇒ (terminate (if false
                 (rec x x)
                 (proc y 1))
             7)

⇒ (terminate 7
             (proc y 1))

⇒ 7
```

Problem 3: Denotational Semantics and Mutation [30 points]

Alyssa P. Hacker has decided to add a new feature to FLK!: the ability to undo store modifications. She introduces a single new form into FLK! called `undo!`:

```
(undo!)   ; undo the last mutation to the store
```

Here is an example of how `undo!` can be used to undo a `cell-set!` using FL! sugar:

```
(let ((c (cell 0)))
   (begin
      (primop cell-set! c 1)
      (undo!)
      (primop cell-ref c)))
```

$\Rightarrow$ *0*

Before Alyssa is called away to testify at a large antitrust trial, she managed to update the Standard Semantics as follows:

$$\begin{aligned} \text{Cmdcont} &= \text{StoreStack} \to \text{Answer} \\ z \in \text{StoreStack} &= \text{Store}^* \end{aligned}$$

$$\mathcal{TL}[\![E]\!] = \mathcal{E}[\![E]\!]\ \textit{empty-env}\ \textit{top-level-cont}\ \textit{empty-store}.[]_{Store}$$

You can assume all of the semantic algebra helper functions that deal in CmdCont have been properly updated. Store operations *same-location?*, *next-location*, *empty-store*, *fetch*, *assign*, *fresh-loc*, and *first-fresh* remain unchanged.

a. [10 points] As she was running out the door, Alyssa mumbled something about the dangers of undoing cell creations using `undo!`. Give an example of a program that would generate a run-time error if Alyssa permits the `cell` form to be undone.

Solution:

```
(let ((c (cell 0)))
   (begin
      (undo!)
      (primop cell-ref c)))
```

b. [10 points] Dissatisfied with the possibility of run-time errors, Alyssa faxes you a revised meaning function for `cell` and assures you that we can prevent undos to cell creations.

$$\begin{aligned} \mathcal{E}[\![\texttt{(primop cell E)}]\!] = \lambda ek\ .\ \mathcal{E}[\![\texttt{E}]\!]\ \text{e}\ (\lambda vz\ .\ & \textbf{matching}\ z \\ & \triangleright s.z_1\ [\![\ (k\ (\text{Location} \mapsto \text{Value}\ \ (\textit{fresh-loc}\ \ s)) \\ & \qquad (\textit{assign-to-all}\ \ (\textit{fresh-loc}\ \ s)\ \ v\ \ z)) \\ & \textbf{endmatching}\ ) \end{aligned}$$

$$\begin{aligned} &\textit{assign-to-all} : \text{Location} \to \text{Storable} \to \text{StoreStack} \to \text{StoreStack} \\ &= \textbf{fix}_{\text{Location} \to \text{Storable} \to \text{StoreStack} \to \text{StoreStack}}\ \lambda f\ .\ (\lambda l \sigma z\ .\ \textbf{matching}\ z \\ &\qquad \triangleright []_{Store}\ [\![\ []_{Store} \\ &\qquad \triangleright s.z_1\ [\![\ (\textit{assign}\ \ l\ \ \sigma\ \ s).\ (f\ l\ \sigma\ z_1) \\ &\qquad \textbf{endmatching}\ ) \end{aligned}$$

Provide a revised meaning function for $\mathcal{E}[\![\texttt{(primop cell-set!}\ \ E_1\ \ E_2\texttt{)}]\!]$.

Solution:

$\mathcal{E}[\![$`(primop cell-set!` E_1 E_2`)`$]\!]$
= λek . $\mathcal{E}[\![E_1]\!]$ e (*test-location* (λl . $\mathcal{E}[\![E_2]\!]$ e (λvz . **matching** z
$\triangleright$ $s.z_1$ $\|$ (k (Unit $\mapsto$Value $unit$) ($assign$ l v s).z)
endmatching)))

c. [10 points] Also provide a revised meaning function for Alyssa's new `undo!` form, $\mathcal{E}[\![$`(undo!)`$]\!]$. Her notes specified that even in the absense of `cell-set!`'s, `undo!` should not produce any errors.

Solution:

$\mathcal{E}[\![$`(primop undo!)`$]\!]$ = λekz . **matching** z
$\triangleright$ $s.[]_{Store}$ $\|$ (k (Unit $\mapsto$Value $unit$) $s.[]_{Store}$)
$\triangleright$ $s.z_1$ $\|$ (k (Unit $\mapsto$Value $unit$) z_1)
endmatching

Problem 4: Denotational Semantics: Control [30 points]

Ben Bitdiddle is now ensconced in a major research university where he's been fooling around with loops in FLK!. Ben has decided to add the following features to FLK!:

$$E ::= \ldots \text{all FLK! forms} \ldots \mid \texttt{(loop } E\texttt{)} \mid \texttt{(exit } E\texttt{)} \mid \texttt{(jump)}$$

Here's an example of how Ben's new loop construct would work:

```
(let ((c (cell 0)))
   (loop
      (begin
         (primop cell-set! c (+ (primop cell-ref c) 1))
         (if (> (cell-ref c) 10)
            (exit (primop cell-ref c))
            (jump)))))
```

$\Rightarrow$ `11`

As one of Ben's grad students, your job is to write the denotational semantics for FLK! with his new looping functionality. Ben has already revised the domain equations for you:

$$\begin{aligned} j \in \text{Jumpcont} &= \text{Cmdcont} \\ x \in \quad \text{Exitcont} &= \text{Expcont} \end{aligned}$$

He's also changed the signature of the $\mathcal{E}$ meaning function so that every expression is evaluated with both a jump and an exit continuation:

$$\mathcal{E} \ : \text{Exp} \rightarrow \text{Environment} \rightarrow \text{Jumpcont} \rightarrow \text{Exitcont} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont}$$

Ben did give you the meaning function clause for $\mathcal{E}[\![\texttt{(jump)}]\!]$ for reference.

$$\mathcal{E}[\![\texttt{(jump)}]\!] = \lambda ejxk \,.\, j$$

a. [10 points] Extend the meaning function in the FLK! Standard Semantics to include $\mathcal{E}[\![\texttt{(loop } E\texttt{)}]\!]$, given the new domains.

Solution:

$$\mathcal{E}[\![\texttt{(loop } E\texttt{)}]\!] = \lambda ejxk \,.\, (\mathbf{fix}_{Cmdcont}\ (\lambda j_1 \,.\, (\mathcal{E}[\![E]\!]\ e\ j_1\ k\ k)))$$

b. [10 points] Define the meaning function clause for $\mathcal{E}[\![\texttt{(exit } E\texttt{)}]\!]$.

Solution:

$$\mathcal{E}[\![\texttt{(exit } E\texttt{)}]\!] = \lambda ejxk \,.\, \mathcal{E}[\![E]\!]\ e\ j\ x\ x$$

c. [10 points] Show that `(loop (jump))` is equivalent to bottom.

Solution:

$$\mathcal{E}[\![\texttt{(loop (jump))}]\!] = \lambda ejxk \,.\, (\mathbf{fix}\ (\lambda j_1 \,.\, (\mathcal{E}[\![(jump)]\!]\ e\ j_1\ k\ k)))$$

$\Rightarrow$

$$\mathcal{E}[\![\texttt{(loop (jump))}]\!] = \lambda ejxk \,.\, (\mathbf{fix}\ (\lambda j_1 \,.\, ((\lambda ejxk \,.\, j)\ e\ j_1\ k\ k)))$$

$\Rightarrow$

$$\mathcal{E}[\![\texttt{(loop (jump))}]\!] = \lambda ejxk \,.\, (\mathbf{fix}\ (\lambda j_1 \,.\, j_1))$$

$\Rightarrow$

$$\mathcal{E}[\![\texttt{(loop (jump))}]\!] = \bot$$