MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

2002 Midterm Solutions

Problem 1: Short Answer [18 points]

Evaluate the following expressions in the given models. If the expression evaluates to an error, say what kind of error it is.

```
(let ((x 1))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 2))
      (f 1))))
```

a. [2 points] static scoping, call by value

Solution: 2

b. [2 points] dynamic scoping, call by value

Solution: 3

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

c. [2 points] static scoping, call by value

Solution: error, `f` is not bound

d. [2 points] dynamic scoping, call by name

Solution: 2

```
(let ((x 0))
  (let ((f (lambda (y) (/ y x))))
    (let ((x 1))
      (set! f (lambda (y) (/ y x))))
    (let ((x 2))
      (f x))))
```

e. [2 points] static scoping, call by value

Solution: 2

f. [2 points] dynamic scoping, call by value

Solution: 1

```
(let ((x (/ 1 0))
      (y 0))
  (let ((z (begin (set! y (+ y 1)) 5)))
    ((lambda (x) (x (x x)))
     (lambda (x) (+ z y z y)))))
```

g. [2 points] static scoping, call by name

Solution: 13

h. [2 points] static scoping, call by value

Solution: error, divide by 0

i. [2 points] static scoping, call by need

Solution: 12

Problem 2: Operational Semantics: [18 points]

Ben Bitdiddle's company, which sells commercial PostFix implementations, has been hard-hit by the Internet stock bust and has sent him off to MIT to bring back new commercializable technology. Ben Bitdiddle has been learning about functional programming, and while he still prefers PostFix, he is intrigued by the notion of currying. He proposes two new PostFix constructs that permit creating and taking apart PostFix procedures. The constructs are called `pack` and `unpack`.

- `pack` expects the first value on the stack to be a number n, and it expects there to be at least n more values on the stack. It packages together the next n values on the stack, $c_n, \ldots, c_1$, as a command sequence $C = (c_1 \ \ldots \ c_n)$ and pushes C on the stack.
- `unpack` expects the first value on the stack to be a command sequence $C = (c_1 \ldots c_n)$. It pushes $c_1, \ldots, c_n, n$ on the stack in that order.

If the preconditions are not met, the operational semantics gets stuck.

`unpack` permits the PostFix stack to contain commands, which was previously impossible. For example, consider the following PostFix program:

$$(N_3 \ \ N_2 \ \ N_1 \ \ \texttt{(add add) exec}) \Rightarrow N_1 + N_2 + N_3$$

We can think of `(add add)` as a procedure of three arguments that adds N_1, N_2, and N_3. Using `unpack` and `pack`, we can write a currying procedure that takes a three-argument procedure, N_1, and N_2, and outputs a procedure that takes one argument N_3 and outputs $N_1 + N_2 + N_3$. The currying procedure is `(unpack 2 add pack)` and it works as follows:

$$(N_2 \ \ N_1 \ \ \texttt{(add add) (unpack 2 add pack) exec}) \Rightarrow (N_2 \ N_1 \text{ add add})$$

Ben's company has built proprietary optimization technology that can convert this command sequence to `(`N_4 `add)`, where $N_4 = N_1 + N_2$. Together, these two innovations promise a remarkable improvement in PostFix efficiency.

a. [5 points] Give a rewrite rule for `unpack`.

Solution:

$$\langle \texttt{unpack} \,.\, Q, \ \ (V_1 \ \ V_2 \ \ \ldots \ \ V_n) \,.\, S \rangle \Rightarrow \langle Q, \ \ n \,.\, V_n \ldots V_1 \,.\, S \rangle \qquad [\textit{unpack}]$$

b. [5 points] Give a rewrite rule for `pack`.

Solution:

$$\langle \texttt{pack} \,.\, Q, \ \ n \,.\, V_n \ldots V_1 \,.\, S \rangle \Rightarrow \langle Q, \ \ (V_1 \ \ V_2 \ \ \ldots \ \ V_n) \,.\, S \rangle \qquad [\textit{pack}]$$

c. [8 points] In addition to performing partial evaluation, Ben would like to be able to reuse its results; after all, procedures that can only be called once are of limited use. Ben proposes to add a restricted form of `dup` to PostFix+{`unpack`, `pack`}; the restricted `dup` may only be used immediately after `pack`. Do all such programs terminate? Argue briefly: give either an energy function or a counterexample.

Solution: PostFix+{`unpack`, `pack`, $\texttt{dup}_{\text{restricted}}$} programs may not terminate. Recall that the canonical non-terminating PostFix+`dup` program is `(dup exec) dup exec`. Notice that `unpack` and `pack` are duals. The following program does not terminate.

```
(unpack pack dup exec) unpack pack dup exec
```

One can also write unrestricted `dup` in terms of restricted `dup`:

```
dup = 1 pack dup unpack pop swap unpack pop
```

Problem 3: Denotational Semantics: [34 points]

Ben Bitdiddle enjoys the convenience of short-circuiting operators and has a proposal for making them even more powerful.

A standard short-circuiting logical operator evaluates only as many of its operands as necessary; it evaluates its arguments in left-to-right order, stopping as soon as it evaluates an argument that determines the value of the entire expression. For instance, if `and` is a short-circuiting operator, then the following program evaluates to `#f` without raising an error:

```
(and #f (= 0 (/ 1 0)))
```

However, reversing the order of the expressions leads to an error:

```
(and (= 0 (/ 1 0)) #f)
```

Ben Bitdiddle reasons that the second expression, too, should evaluate to `#f`. After all, one of the operands evaluates to `#f`, and that determines the value of the entire expression. He proposes a very-short-circuiting operator `nd-and` (*non-deterministic and*) such that if either operand evaluates to false, then only that operand is evaluated; otherwise, both operands are evaluated. His goals are:

- The expression errs or infinite-loops only if at least one of the operands does, and the other expression does not evaluate to `#f`. (Hint: infinite loops, errors, and concurrency are not the main point of this problem.)
- The value of the entire expression is the `and` of all the visibly evaluated operands, where a visibly executed operand is one whose side effects have been performed on the resulting store.
- The entire expression evaluates to `#t` if and only if both operands are visibly evaluated (because both operands must be evaluated to achieve that result).
- The entire expression evaluates to `#f` if and only if exactly one expression is visibly evaluated.

Alyssa P. Hacker does not believe Ben's goals are achievable. She says she can satisfy the first two goals plus either of the last two goals, but not all four goals simultaneously.

a. [6 points] Informally describe the operational semantics for one of the possibilities for `nd-and` that satisfies Alyssa's claim.

Solution: There are multiple solutions to this problem. Here is one of them.

Let s_0 be the initial store.

Evaluate E_1 in s_0, giving $\langle v, s_1 \rangle$. If v is false, return $\langle v, s_1 \rangle$.

Otherwise, evaluate E_2 in s_0, giving $\langle v, s_2 \rangle$. If v is false, return $\langle v, s_2 \rangle$.

Otherwise, evaluate E_2 in s_1, giving $\langle v, s_3 \rangle$. Return $\langle v, s_3 \rangle$.

(This might be false, even though both operands have been evaluated.)

b. [8 points] What is $\mathcal{E}[\![(\texttt{nd-and}\ E_1\ E_2)]\!]$ for the version of `nd-and` that you described above?

Solution:

$$
\begin{array}{l}
\mathcal{E}[\![(\texttt{nd-and}\ E_1\ E_2)]\!] \\
= \lambda e\, k s_0\ .\ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_1\ .\ \textbf{if}\ \neg v_1 \\
\qquad\qquad\qquad\qquad \textbf{then}\ (k\ \text{false}) \\
\qquad\qquad\qquad\qquad \textbf{else}\ \mathcal{E}[\![E_2]\!]\ e\ (\lambda v_2\ .\ \textbf{if}\ \neg v_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (k\ \text{false}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathcal{E}[\![E_1]\!]\ e\ k) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s_0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s_0
\end{array}
$$

Alternately, with explicit stores:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(nd-and } E_1 \ E_2\texttt{)}]\!] \\ = \lambda e k s_0 \ . \ \mathcal{E}[\![E_1]\!]\ e\ (\lambda v_1\, s_1 \ . \ \textbf{if } \neg v_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } (k \ \text{false}\ s_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathcal{E}[\![E_2]\!]\ e\ (\lambda v_2\, s_2 \ . \ \textbf{if } \neg v_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } (k \ \text{false}\ s_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathcal{E}[\![E_1]\!]\ e\ k\ s_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad s_0) \\ \qquad\qquad\qquad\qquad\qquad\qquad s_0\end{array}$$

c. [4 points] Can `nd-or` (nondeterministic or) be defined in terms of `nd-and`? Explain briefly.

Solution: Yes.

$$\mathcal{D}_{\text{exp}}[\![\texttt{(nd-or } E_1 \ E_2\texttt{)}]\!] \quad = \quad \texttt{(not (nd-and (not } E_1\texttt{) (not } E_2\texttt{)))}$$

Because `not` preserves nontermination and errors, the semantics are as desired: if either E_1 or E_2 evaluates to true, the other is not evaluated.

d. [3 points] What does the following FLAVAR! program evaluate to?

```
(let ((a 2))
  (let ((and-result (nd-and (= a 3)
                            (begin (set! a 3) #t))))
    (list and-result a)))
```

Solution: E_1 evaluates to false, and E_2 evaluates to true. The entire expression evaluates to `(#f 2)`.

e. [3 points] What does the following FLAVAR! program evaluate to?

```
(let ((a 2))
  (let ((and-result (nd-and (= a 2)
                            (begin (set! a 3) #t))))
    (list and-result a)))
```

Solution: E_1 in isolation evaluates to true, and E_2 evaluates to true. The entire expression evaluates to either `(#t 3)` or `(#f 3)`, depending on the order of evaluation.

f. [6 points] Demonstrate that Alyssa's assertion is correct. Given your semantics for `nd-and`, write an `nd-and` expression that fails one of the last two constraints. The expression should either definitely evaluate to `#t`, but with the side effects of just one of its arguments; or it should definitely evaluate to `#f`, but with the side effects of both arguments.

Solution: This expression evaluates to false, but evaluates both arguments:

```
(let ((a #t) (b #t))
  (nd-and (begin (set! b #f) a)
          (begin (set! a #f) b)))
```

g. [4 points] Suggest a restriction (to FLAVAR!, FLK!, or `nd-and`) that achieves all of Ben's goals.

Solution: Disallow uses of `set!`.

Problem 4: Control [30 points]

After hearing that Ben Bitdiddle's MIT experience led him to experiment with currying (Problem 2), the president of Ben's company exclaimed, "I won't be caught selling buggy whips, curry combs, or other horse products in the modern economy!" and sent Ben off to New Jersey to learn some more practical programming constructs.

Ben noted that FLK! is missing the while loop, which is standard in other languages, and reasons that adding it will reduce programmers' resistance to FLK!.

Ben proposes three new constructs — `while`, `continue`, and `break` — to ensure that C programmers feel at home programming in FLAVAR!. The command (`while` E_{cond} E_{body} E_{final}) behaves as follows. If E_{cond} is true, then evaluate E_{body} and loop back to re-evaluate the entire `while` form (starting with E_{cond} again). If E_{cond} is false, then the value of the entire `while` expression is the result of evaluating E_{final}.

Within E_{body}, (`continue`) preempts execution of the smallest enclosing E_{body} and returns to the top of that loop.

Finally, (`break` E_3) forces the entire `while` expression to terminate with the value E_3 (without evaluating E_{final}).

Consider the following procedure:

```
(define f
  (lambda (xval)
    (let ((x (cell xval)))
       (while (begin (cell-set! x (+ (cell-ref x) 1)) (< (cell-ref x) 0))
              (begin (cell-set! x (+ (cell-ref x) 1))
                     (if (< (cell-ref x) 0)
                         (continue)
                         (break 42)))
              (- (cell-ref! x) 1)))))
```

Evaluation proceeds as follows:

(`f -10`) ⇒ `42`
(`f -11`) ⇒ `-1`

In order to provide a meaning for the new commands, we must change the meaning function $\mathcal{E}$ and add a new domain:

$\mathcal{E}$: Exp → Environment → Expcont → ContCont → BreakCont → Cmdcont
c ∈ ContCont = Expcont
b ∈ BreakCont = Expcont

a. [10 points] What is $\mathcal{E}[\![$(`while` E_{cond} E_{body} E_{final})$]\!]$?

Solution: Here is a version that re-evaluates the condition after (`continue`), and where (`break`) or (`continue`) in E_{cond} break out of the `while` form to which E_{cond} belongs:

$$\begin{aligned}&\mathcal{E}[\![\texttt{(while } E_{\text{cond}}\ E_{\text{body}}\ E_{\text{final}}\texttt{)}]\!]\\ &= \lambda e k c b\,.\ \mathbf{fix}_{\text{Cmdcont}}\,\lambda l\,.\ \mathcal{E}[\![E_{\text{cond}}]\!]\,e\,(\lambda v\,.\ \mathbf{if}\ v\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathcal{E}[\![E_{\text{body}}]\!]\,e\,(\lambda v\,.\ l)\,(\lambda v\,.\ l)\,k\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathcal{E}[\![E_{\text{final}}]\!]\,e\,k\,c\,b)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda v\,.\ l)\,k\end{aligned}$$

Here is a version that does not re-evaluate the condition after (`continue`), and where (`break`) or (`continue`) in E_{cond} break out of a `while` form that encloses the one to which E_{cond}belongs:

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(while } E_{\text{cond}} \ E_{\text{body}} \ E_{\text{final}}\texttt{)}]\!] \\
&= \lambda e k c b \,.\ \mathbf{fix}_{\text{Cmdcont}}\, \lambda l \,.\ \mathcal{E}[\![E_{\text{cond}}]\!]\, e\, (\lambda v \,.\ \mathbf{if}\ v \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathbf{fix}_{\text{Cmdcont}}\, (\lambda j \,.\ \mathcal{E}[\![E_{\text{body}}]\!]\, e\, (\lambda v \,.\ l)\, (\lambda v \,.\ j)\, k) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathcal{E}[\![E_{\text{final}}]\!]\, e\, k\, c\, b) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c\, b
\end{aligned}$$

b. [10 points] What is $\mathcal{E}[\![\texttt{(continue)}]\!]$?

Solution: $\mathcal{E}[\![\texttt{(continue)}]\!] = \lambda e k c b \,.\ (c\ \texttt{unit})$

c. [10 points] What is $\mathcal{E}[\![\texttt{(break } E\texttt{)}]\!]$?

Solution: $\mathcal{E}[\![\texttt{(break } E\texttt{)}]\!] = \lambda e k c b \,.\ \mathcal{E}[\![E]\!]\, e\, b\, c\, b$