

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

2002 Midterm

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.

Contents

Problem 1: Short Answer [18 points]	2
Problem 2: Operational Semantics: [18 points]	3
Problem 3: Denotational Semantics: [34 points]	4
Problem 4: Control [30 points]	6
Appendix A: PostFix Grammar	7
Appendix B: PostFix SOS from Chapter 3	8
Appendix C: Standard Semantics of FLK!	9

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

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

- c. [2 points] static scoping, call by value
 - d. [2 points] dynamic scoping, call by name
-

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

```
(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
- h. [2 points] static scoping, call by value
- i. [2 points] static scoping, call by need

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, \dots, c_1 , as a command sequence $C = (c_1 \dots c_n)$ and pushes C on the stack.
- `unpack` expects the first value on the stack to be a command sequence $C = (c_1 \dots c_n)$. It pushes c_1, \dots, 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:

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

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:

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

Ben's company has built proprietary optimization technology that can convert this command sequence to `(N4 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`.
- b. [5 points] Give a rewrite rule for `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.

Problem 3: Denotational Semantics: [34 points]

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

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.

- [6 points] Informally describe the operational semantics for one of the possibilities for `nd-and` that satisfies Alyssa's claim.
- [8 points] What is $\mathcal{E}[(\text{nd-and } E_1 E_2)]$ for the version of `nd-and` that you described above?
- [4 points] Can `nd-or` (nondeterministic or) be defined in terms of `nd-and`? Explain briefly.
- [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)))
```

- [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)))
```

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

Problem 4: Control [30 points]

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

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:

$$\begin{aligned} \mathcal{E} &: \text{Exp} \rightarrow \text{Environment} \rightarrow \text{Expcont} \rightarrow \text{ContCont} \rightarrow \text{BreakCont} \rightarrow \text{Cmdcont} \\ c &\in \text{ContCont} = \text{Expcont} \\ b &\in \text{BreakCont} = \text{Expcont} \end{aligned}$$

- [14 points] What is $\mathcal{E}[(\text{while } E_{\text{cond}} E_{\text{body}} E_{\text{final}})]$?
- [8 points] What is $\mathcal{E}[(\text{continue})]$?
- [8 points] What is $\mathcal{E}[(\text{break } E)]$?

Appendix A: PostFix Grammar

$P \in$ Program
$Q \in$ Commands
$C \in$ Command
$A \in$ Arithmetic-operator = {add, sub, mul, div}
$R \in$ Relational-operator = {lt, eq, gt}
$N \in$ Intlit = {..., -2, -1, 0, 1, 2, ...}
$P ::= (Q)$ [Program]
$Q ::= C^*$ [Command-sequence]
$C ::= N$ [Integer-literal]
pop [Pop]
swap [Swap]
A [Arithmetic-op]
R [Relational-op]
sel [Select]
exec [Execute]
(Q) [Executable-sequence]

Figure 1: The S-Expression Grammar for PostFix

Appendix B: PostFix SOS from Chapter 3

$\mathcal{C}, \mathcal{F}, \mathcal{I}$ and \mathcal{O} for our PostFix SOS are given by:

$\mathcal{C} = \text{Commands} \times \text{Stack}$

$\mathcal{F} = \{\langle _ \rangle_{\text{Command}}\} \times \text{Stack}$

$\mathcal{I} : \text{Program} \rightarrow \mathcal{C}$
 $= \lambda P . \mathbf{matching} P$
 $\quad \triangleright \langle Q \rangle \parallel \langle Q, \langle _ \rangle_{\text{Value}} \rangle$
 $\quad \mathbf{endmatching}$

$\mathcal{O} : \mathcal{F} \rightarrow \text{Answer}$
 $= \lambda \langle _ \rangle_{\text{Command}}, S . \mathbf{matching} S$
 $\quad \triangleright V . S' \parallel (\text{Value} \mapsto \text{Answer } V)$
 $\quad \triangleright \langle _ \rangle \parallel (\text{Error} \mapsto \text{Answer } \text{error})$
 $\quad \mathbf{endmatching}$

$\langle N . Q, S \rangle \Rightarrow \langle Q, N . S \rangle$	[numeral]
$\langle \langle Q_{exec} \rangle . Q_{rest}, S \rangle \Rightarrow \langle Q_{rest}, Q_{exec} . S \rangle$	[executable]
$\langle \text{pop} . Q, V_{top} . S \rangle \Rightarrow \langle Q, S \rangle$	[pop]
$\langle \text{swap} . Q, V_1 . V_2 . S \rangle \Rightarrow \langle Q, V_2 . V_1 . S \rangle$	[swap]
$\langle \text{sel} . Q_{rest}, V_{false} . V_{true} . 0 . S \rangle \Rightarrow \langle Q_{rest}, V_{false} . S \rangle$	[sel-false]
$\langle \text{sel} . Q_{rest}, V_{false} . V_{true} . N_{test} . S \rangle \Rightarrow \langle Q_{rest}, V_{true} . S \rangle$ where $N_{test} \neq 0$	[sel-true]
$\langle \text{exec} . Q_{rest}, Q_{exec} . S \rangle \Rightarrow \langle Q_{exec} @ Q_{rest}, S \rangle$	[execute]
$\langle A . Q, N_1 . N_2 . S \rangle \Rightarrow \langle Q, N_{result} . S \rangle$ where $N_{result} \equiv (\text{calculate } A \ N_2 \ N_1)$ and $\neg((A \equiv \text{div}) \wedge (N_1 \equiv 0))$	[arithop]
$\langle R . Q, N_1 . N_2 . S \rangle \Rightarrow \langle Q, 1 . S \rangle$ where $(\text{compare } R \ N_2 \ N_1)$	[relop-true]
$\langle R . Q, N_1 . N_2 . S \rangle \Rightarrow \langle Q, 0 . S \rangle$ where $\neg(\text{compare } R \ N_2 \ N_1)$	[relop-false]

Figure 2: Rewrite rules defining the transition relation for PostFix.

Appendix C: Standard Semantics of FLK!

$v \in \text{Value} = \text{Unit} + \text{Bool} + \text{Int} + \text{Sym} + \text{Pair} + \text{Procedure} + \text{Location}$ $k \in \text{Expcont} = \text{Value} \rightarrow \text{Cmdcont}$ $\gamma \in \text{Cmdcont} = \text{Store} \rightarrow \text{Expressible}$ $\text{Expressible} = (\text{Value} + \text{Error})_{\perp}$ $\text{Error} = \text{Sym}$ $p \in \text{Procedure} = \text{Denotable} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont}$ $d \in \text{Denotable} = \text{Value}$ $e \in \text{Environment} = \text{Identifier} \rightarrow \text{Binding}$ $\beta \in \text{Binding} = (\text{Denotable} + \text{Unbound})_{\perp}$ $\text{Unbound} = \{\text{unbound}\}$ $s \in \text{Store} = \text{Location} \rightarrow \text{Assignment}$ $l \in \text{Location} = \text{Nat}$ $\alpha \in \text{Assignment} = (\text{Storable} + \text{Unassigned})_{\perp}$ $\sigma \in \text{Storable} = \text{Value}$ $\text{Unassigned} = \{\text{unassigned}\}$ $\text{top-level-cont} : \text{Expcont}$ $= \lambda v . \lambda s . (\text{Value} \mapsto \text{Expressible } v)$ $\text{error-cont} : \text{Error} \rightarrow \text{Cmdcont}$ $= \lambda y . \lambda s . (\text{Error} \mapsto \text{Expressible } y)$ $\text{empty-env} : \text{Environment} = \lambda I . (\text{Unbound} \mapsto \text{Binding } \text{unbound})$ $\text{test-boolean} : (\text{Bool} \rightarrow \text{Cmdcont}) \rightarrow \text{Expcont}$ $= \lambda f . (\lambda v . \mathbf{matching } v$ $\quad \triangleright (\text{Bool} \mapsto \text{Value } b) \parallel (f b)$ $\quad \triangleright \mathbf{else } (\text{error-cont } \text{non-boolean})$ $\quad \mathbf{endmatching })$ Similarly for: $\text{test-procedure} : (\text{Procedure} \rightarrow \text{Cmdcont}) \rightarrow \text{Expcont}$ $\text{test-location} : (\text{Location} \rightarrow \text{Cmdcont}) \rightarrow \text{Expcont}$ etc. $\text{ensure-bound} : \text{Binding} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont}$ $= \lambda \beta k . \mathbf{matching } \beta$ $\quad \triangleright (\text{Denotable} \mapsto \text{Binding } v) \parallel (k v)$ $\quad \triangleright (\text{Unbound} \mapsto \text{Binding } \text{unbound}) \parallel (\text{error-cont } \text{unbound-variable})$ $\quad \mathbf{endmatching }$ Similarly for: $\text{ensure-assigned} : \text{Assignment} \rightarrow \text{Expcont} \rightarrow \text{Cmdcont}$
--

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 = λs . (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
lookup : Environment → Identifier → Binding = λe I . (e I)

```

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

