**Principles of Programming Languages (CS329)** **Midterm**
**Computer Science and Engineering** **2003-09-29**
**Indian Institute of Technology Bombay** **15:00–17:00**

NAME _____ ROLL _____

This midterm has 2 pages. Write your answer clearly in the spaces provided. Do not write in the boxes meant for scoring your work. Do not attach rough work. Use the marks alongside each question for time management. Provide 1–2 sentences of informal justification to qualify for partial credit in case your final answer is wrong. You can use the FWH book and your own class notes only.

1. We attempt to write a generic `loop` construct in Scheme

```
(define (loop body)
  (let ((r 'junk))
    (begin
      (call-with-current-continuation (lambda (c) (set! r c)))
      body
      (display "next iteration\n")
      (r 'junk))))
```

   so that we can call it as follows:

```
(define counter 1)
(loop (begin
        (display (set! counter (1+ counter)))
        (newline) ) )
```

   (a) What will the above code print when evaluated?

| | 1 |
|---|---|

   (b) How can you fix it to give the desired response?

| | 1 |
|---|---|

2. If we have mutable store, we can make mutable objects. E.g., we can create a `point` object with two methods `get` and `set`, which will make the following code

```
let point-class = ...
  let point-instance = (point-class 'new 3)
    (begin
      (point-instance 'set 5)
      (point-instance 'get) )
```

   evaluate to 5. If we do not have mutable store, any mutating method (like `set`) must return a new object, and the code above must change to

```
let point-class = ...
  let point-instance = (point-class 'new 3)
    let another-point-instance =  (point-instance 'set 5)
      (another-point-instance 'get)
```

   Use `rec` to write the second (non-mutating) version of `point-class` (i.e., complete the "`...`"). Your solution should indicate a generic recipe and not be specific to this particular class or instance. Limited mix-up of Scheme and FLK syntax will be tolerated. You should not need any store support.

| | 2 |
|---|---|

3. The new programming language Darjeeling from Moon Microsystems gives the programmer exception constructs, but with "retry" semantics.

   Informally, this means that a `throw` statement executes a handler, and then the `try` block is reevaluated. In case of multiple nested `try`s, that `try` block associated with the active handler is restarted. Retrying is repeated until the `try` block executes normally without invoking a `throw`.

   For simplicity, assume that handler names are looked up using static (lexical) scoping. Give a translation of Darjeeling's exception construct to Scheme. You may use `let`, `lambda`, calls, `callcc`, `begin`, and `set!`.

| | 3 |
|---|---|

4. Consider two recursive function applications, written using `rec` in FLK (no mutable store):

   `(call (rec foo` $E_{p1}$`)` $E_{a1}$`)` and `(call (rec bar` $E_{p2}$`)` $E_{a2}$`)`

   Ace FLK programmer Games Jostling appreciates that (thanks to recursion) one or both of these expressions may "hang" the lambda engine (i.e., trap it in an infinite reduction sequence). Jostling wants to design the new FLK construct

```
(jostle (call (rec foo ...) ...)
        (call (rec bar ...) ...)  )
```

   with the following informal semantics

   - If both expressions hang, `jostle` hangs the lambda engine.
   - If exactly one expression reduces to normal form in a finite number of steps, `jostle` returns that normal form and does not fall prey to the other unending reduction.
   - If both expressions reduce to normal forms in a finite number of steps, one of the two normal form expressions is returned arbitrarily.

   Your job is to help Jostling write a translation of the `jostle` construct to lambda expressions. You may use any tricks from lambda calculus, including (possibly hacked up versions of) $Y$-combinators, Church numerals, etc.

| | 4 |
|---|---|

5. This question concerns standard semantics for FLK and all its extensions discussed in class.

   (a) Give the shortest possible example code $E$ which establishes that

   $$(\mathcal{E}\ [\![\mathtt{E}]\!]\ u\ k_1)$$

   and

   $$\Big(k_1\ (\mathcal{E}\ [\![\mathtt{E}]\!]\ u\ (\lambda\ x\ x))\Big)$$

   are not equivalent.

| | 2 |
|---|---|

   (b) What condition must $E$ satisfy so that the two forms above can be guaranteed to evaluate to the same value?

| | 2 |
|---|---|

**Total: 15**