Programming Languages (CS329) **Quiz**
Computer Science and Engineering **2002-10-08 14:15–17:15**
Indian Institute of Technology Bombay **Open book/notes**

NAME ______________________________ ROLL ____________________

This exam consists of pages 1 through 6. Credit for each question is marked up alongside and the total is given at the end. Use these credits for time management.

You may use the FWH book, your own notes, and handouts posted on the course Web pages in the last three years. You cannot use notes written by another student which you have copied, unless you wrote the notes in a fair collaboration.

If a question is ambiguous, make reasonable assumptions and write them down. It is advisable to provide some verbal intuition, because this may earn you partial credit in case the final answer is wrong. Write your answers only in the spaces provided. Carry out any rough work on separate sheets of paper; do not attach rough sheets. Do not write inside the boxes meant for entering scores.

1. Write down the value of the following expression

```
let x = 1  n = 10
  let add-x = (proc n (+ x n))
    let x = 100  n = 1000
      (call add-x n)
```

   under applicative order and

   (a) Static scoping

   $\boxed{}\boxed{1}$

   (b) Dynamic scoping

   $\boxed{}\boxed{2}$

2. Recall the definitions of `PAIR`, `LEFT` and `RIGHT`.

   (a) What does this expression evaluate to?

   `(LEFT (rec camel (PAIR` $\bar{4}$ `(LEFT camel))) )`

   $\boxed{}\boxed{1}$

   (b) What does this expression evaluate to?

   `(RIGHT (rec camel (PAIR` $\bar{4}$ `(LEFT camel))) )`

   $\boxed{}\boxed{2}$

   (c) The following Scheme code is first entered at the top level:

```
(define (left p) (p (lambda (x y) x)))
(define (right p) (p (lambda (x y) y)))
(define (mystery so) (so 4 (lambda (si) (mystery si))))
```

   Does the Scheme interpreter go into an infinite computation on defining these functions?

   $\boxed{}\boxed{1}$

Write down the value of the following expressions, or assert that the interpreter will go into an infinite computation.

```
(left mystery)
```

1

```
(right mystery)
```

1

```
(left (right mystery))
```

1

```
(left (right (right mystery)))
```

1

Write down in one sentence what `mystery` represents.

1

3. The length of a list can be found using this Scheme function:

```
(define (length-rec lst) (if (null? lst ) 0 (+ 1 (length-rec (cdr lst)))))
```

Convert this to CPS by completing the following template:

```
(define (length-cps lst cont) ...)
```

Note that `null?` and `cdr` are primitive functions. 3

4. Complete the following CPS conversion of the `callcc` construct from FLK to FLTR.

$\mathcal{C}$`[(callcc` E`)] = (proc` k_{ret} $\mathcal{C}$`[`E`] (proc f` ____________________`) )`

3

Verify that your proposed solution works for the following code (show the important steps).

```
(+ 1 (callcc (proc (c) (* 2 (call c 3))) )  )
```

	2

5. We would like to augment the constructs `(loop` *E*`)` and `(break` *E*`)`, discussed in the midterm exam, with the new construct `(continue)`, which serves the same purpose as in C/C++/Java: it skips the computation for the rest of the current iteration and continues the loop from the next iteration. For example, the following code

   ```
   let px = (new 0)
     (loop (begin
       (write px (+ 1 (read px)))
       (if (> (read px) 5) (break) #f)
       (if (= (read px) 3) (continue))
       (display (read px))
       (display " ")  )  )
   ```

   results in the following output:

   ```
   1 2 4 5
   ```

   Give a general desugaring technique for code containing these three constructs `(loop` *E*`)`, `(break` *E*`)` and `(continue)`, using `callcc` but not using `label` or `jump`. You may assume that the desugaring routine can maintain a stack, and/or generate fresh identifiers if necessary. (Note: the midterm exam problem asked for the eval function $\mathcal{E}$, whereas this problem is about desugaring.)

	6

6. In C++, if

   (a) a method `foo` is not declared `virtual` in the base class `Base`,

   (b) `Ext` extends `Base` and overrides `foo`,

   (c) a pointer to an `Ext` object is cast to a `Base` pointer, and

   (d) the `Base` pointer is used to invoke method `foo`,

   then the method to be invoked is the one defined in the `Base` object.

   We can simulate this in FLOP using the following constructs.

$$\begin{array}{lll} E & \texttt{::=} & \ldots \\ & | & (\texttt{extend}\ \mathtt{I}_{\text{base}}\ E_{\text{base}}\ E_{\text{ext}}) \\ & | & (\texttt{cast}\ \mathtt{I}_{\text{up}}\ E_{\text{obj}}) \end{array}$$

   The `extend` construct behaves the same way as the `object` construct discussed in class, except that it gives precedence to methods found in E_{ext} over those found in E_{base}, and it "remembers" that E_{base} was called $\mathtt{I}_{\text{base}}$ and then extended. The `cast` construct will first check that a base object called $\mathtt{I}_{\text{up}}$ was earlier extended to form E_{obj} (otherwise there is an error) and then return a copy of the named base object. (Note that if all object state is kept in the store, we can still invoke methods in the clone of the base object to modify the state of the extended object.)

   Write denotational semantics (no need for standard semantics with continuations) for these new constructs.

	5

7. Recall the exception-handling syntax in FLK:

$$\begin{array}{rcl} E & \texttt{::=} & \ldots \\ & | & \texttt{(catch (}\texttt{I}_x\ \texttt{(}\texttt{I}_a\texttt{)}\ E_h\texttt{)}\ E_b\texttt{)} \\ & | & \texttt{(throw }\texttt{I}_x\ E_a\texttt{)} \end{array}$$

We define standard semantics for these constructs as follows:

$$\begin{array}{rcl} w \in \mathit{HEnv} & = & \mathrm{Id} \rightarrow \mathit{Handler} \\ \mathit{Handler} & = & \mathit{ExpVal} \rightarrow \mathit{HEnv} \rightarrow \mathit{Store} \rightarrow \mathit{ExpVal} \\ \mathcal{E}[\texttt{(catch (}\texttt{I}_x\ \texttt{(}\texttt{I}_a\texttt{)}\ E_h\texttt{)}\ E_b\texttt{)}] & = & \lambda u_0\, \lambda w_0\, \lambda k_0\, \lambda s_0\,. \end{array}$$

$$\left(\mathcal{E}\,[E_b]\, u_0 \begin{array}{|c|}\hline \texttt{I}_x \rightarrow \lambda e_a \lambda w_1 \lambda s_1 \left(\mathcal{E}[E_h] \begin{array}{|c|}\hline \texttt{I}_a \rightarrow e_a \\ \hline u_0 \\ \hline \end{array} \blacksquare\blacksquare\blacksquare \right) \\ \hline w_0 \\ \hline \end{array}\ k_0\, s_0 \right)$$

(a) Complete the three blanks marked by "■" so as to implement "termination semantics." Note that the blanks are not necessarily "to scale."

| | 3 |

(b) With standard semantics as given above, what will be the value of the following expression?

```
(* 3 (let div = (proc (x y) (if (= 0 y)
                                (throw overflow x)
                                (/ x y)  )  )
       (catch (overflow (num) (+ 1000 num))
         (+ 5 (div 13 0))  )   )    )
```

$\boxed{}\boxed{2}$

(c) Suppose we want "exit semantics," by which we mean that throwing *any* exception using the expression `(throw I`$_x$ E_a`)` binds E_a to the corresponding handler paramter (`I`$_a$), evaluates the corresponding handler body E_h, and terminates the *whole program* with the value of E_h. Complete the three blanks marked by "■" again to satisfy the new spec.

$\boxed{}\boxed{2}$

(d) What would be the value of the expression above under "exit semantics?"

$\boxed{}\boxed{2}$

(e) Under termination semantics, what would be the value of the following expression?

```
(catch (err (y) (+ y 200))
  (let f = (proc (x) (+ (throw err x) 1000))
    (catch (err (z) (+ z 500))
      (call f 4) ) ) )
```

$\boxed{}\boxed{3}$

Total: 43