Programming Languages (CS329) **Midterm exam**
Computer Science and Engineering **2002-09-18 09:30–11:30**
Indian Institute of Technology Bombay **Open book/notes**

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. It is advisable to provide some verbal intuition, because this may earn you partial credit in case the final answer is wrong.

You can use the FWH book, your own notes, and notes 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.

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. Suppose $\mathtt{I}_1$ and $\mathtt{I}_2$ are different identifiers, and $\mathtt{I}_2 \notin FV[E_1]$.

   (a) Give an example of E_1, E_2 and E_3 such that

   $$[E_1/\mathtt{I}_1]\Big([E_2/\mathtt{I}_2]\,E_3\Big) \;\not\equiv\; [E_2/\mathtt{I}_2]\Big([E_1/\mathtt{I}_1]\,E_3\Big).$$

	2

   (b) Although trivial commutativity does not hold, a minor adjustment to the substitution will make things work out. Fill in the blank below.

   $$[E_1/\mathtt{I}_1]\Big([E_2/\mathtt{I}_2]\,E_3\Big) \;\equiv\; \Big[(\rule{1.5em}{0.4pt}\,E_2)/\mathtt{I}_2\Big]\Big([E_1/\mathtt{I}_1]E_3\Big)$$

	3

   (c) Prove by induction that your choice works.

	3

2. (a) Complete this design of the integer decrement (`DEC`) function which, given a Church numeral $\bar{n}$ as input, outputs $\overline{n-1}$. Here `ID` is the identity function, `(λ y y)`.

```
λn . IF (ZERO? n) 0̄
          (n
            (λ x (IF (x ID FALSE) 0̄ (INC x)) )
            _______________ )
```

	2

(b) Explain how your choice works.

	2

3. What is the result of evaluating the following expression using FLK with dynamic scope?

```
let a = 1
  let f = (proc b (+ a b))
    (call f (let a = (call f 20)
               (call f 300))  )
```

	3

4. In class we considered only the single-binding `let` construct. We can easily extend this to a multi-binding `let` construct, provided no right-hand-side uses any of the identifiers to be bound. In other words, no $\mathtt{I}_j$ ($j = 1, \ldots, n$) is allowed to occur free in any E_k ($k = 1, \ldots, n$) in the following code:

```
let I_1 = E_1  I_2 = E_2  ...  I_n = E_n
  E_0
```

We will also assume the availability of multi-argument `procs`, written as `(proc (x y)...)`, with the understanding that a multi-binding `let` can be desugared into a multi-argument `proc`.

While describing Y and `rec`, we have only considered a single recursive function which calls itself. Real languages support *mutual* recursion between a set of functions, which, in FLK style, may be expressed as

```
letrec f = (proc x ...(call g ...)...) g = (proc y ...(call f ...)...)
  ;; use f and g here
```

Desugar the `letrec` construct using `rec` and multi-binding `lets`, by completing the template given below, where $I_c, I_s \notin \cup_{0 \leq j \leq n} FV[E_j]$.

```
𝒟[(letrec I_1 = E_1  ...  I_n = E_n  E_0)] =
    (call (rec I_c (proc I_s
                       (let I_1=(call I_c __________) ... I_n=(call I_c __________)
                         (call __ 𝒟[E_1]  ...  𝒟[E_n]) )
                   )
          ) (proc (I_1 ... I_n) 𝒟[__]) )
```

(We have left around some `lets` for readability, but these are easily removed via additional desugaring.)

	6

5. What is potentially dangerous about the following C++ code fragment? (Hint: it has got to do with virtual methods.)

```
class istream {
 public:
  istream() { }
  ~istream() { }
  virtual int getChar() =0;
};
class ifstream : public istream {
 protected:
  int fileDesc;
 public:
  ifstream(const char * fileName);
  ~ifstream();
  int getChar();
};
```

```
ifstream::ifstream(const char * fn) {
  fileDesc = open(fn, O\_RDONLY);
}
ifstream::~ifstream() {
  close(fileDesc);
}
int ifstream::getChar() {
  // getChar on files
  // implemented here
}
```

Suppose I construct a `ifstream` object (which results in a file descriptor being allocated), and cast an `istream` pointer to this object. I can use the `istream` pointer to invoke the `getChar` method on the `ifstream` object, because `getChar` is virtual. Afterwards, if I `delete` the `istream` pointer (instead of `delete`ing the `ifstream` pointer, because the destructor `istream` is not virtual, the file descriptor will not be released, resulting in a resource leak. See `Delete.cpp` for an example you can play with.

	3

6. The Fibonacci function can be written in FLK in the following manner:

```
(rec fib (proc n
           (if (< n 2) 1
             (+ (call fib (- n 1)) (call fib (- n 2)))  )  )   )
```

Convert the above code to continuation passing style by completing this template:

```
(rec fib-cps (proc (n k) ...)
```

```
(rec fib-cps
  (proc (n k)
    (if (< n 2) (call k 1)          ;; (< n 2) is primitive
      (call fib-cps (- n 1)         ;; (- n 1) is primitive
                   (proc (v1)
                     (call fib-cps (- n 2)
                                  (proc (v2) (call k (+ v1 v2)))
                     ) )
      ) ;call
    ) ;if
  ) ;proc
)
```

	4

7. Here is a session from the MIT Scheme interpreter:

```
1 ]=> (define bump (lambda (x) (begin (display x) x)))
;Value: bump
1 ]=> (* (bump 5) (bump 6))
65
;Value: 30
```

Here are some standard semantics for FLK (no *Store*) to help you. Scheme is similar.

$$\begin{array}{rcl}
\mathit{Cont} & = & \mathit{ExpVal} \rightarrow \mathit{ExpVal} \\
\mathit{Proc} & = & \mathit{ExpVal} \rightarrow \mathit{Cont} \rightarrow \mathit{ExpVal} \\
\mathcal{E} & : & \text{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Cont} \rightarrow \mathit{ExpVal} \\
\mathcal{E}[\![\texttt{(callcc } E\texttt{)}]\!] & = & \lambda u\, \lambda k \left(\mathcal{E}\,[\![E]\!]\, u\, \underline{\lambda p \left(p\, \underline{\lambda d\, \lambda j\, (k\ d)}\ k\right)}\right) \\
\mathcal{E}[\![\texttt{(proc I } E\texttt{)}]\!] & = & \lambda u_0\, \lambda k_0 \left(k_0 \quad \underline{\lambda e \lambda k_1 \left(\mathcal{E}\,[\![E]\!] \boxed{\begin{array}{c}\texttt{I} \rightarrow e \\ \hline u_0\end{array}}\ k_1\right)}\right)
\end{array}$$

What is the result of evaluating the following Scheme code?

(a) `(+ 1 (callcc (lambda (c) (/ 2 (* (c 3) (c 4)) ) )  )  )`

The answer is $1 + 4 = 5$, because '`*`' seems to be evaluated right-associatively in Scheme.

$\boxed{\quad}\boxed{3}$

(b) `(+ 1 (callcc (lambda (c) (* 2 (c (c 10)))) )  )`

The answer is $1 + 10 = 11$. Which `c` is/are involved? Try modifying the code to find out.

$\boxed{\quad}\boxed{3}$

8. We wish to add the following looping construct to FLK!:

$$E \ \ \texttt{::=}\ \ \ldots \mid \texttt{(loop } E\texttt{)} \mid \texttt{(break } E\texttt{)}$$

`(loop` E_1`)` evaluates E_1 repeatedly for ever. `(break` E_2`)` ends the nearest lexically enclosing loop with the return value of E_2.

E.g., the following FLFL code (when translated to FLK!) will evaluate to $1+2+3+4+5 = 15$:

```
let s = 0, i = 0
  (loop
    (begin
      (:= s (+ s i))
      (if (> i 5) (break s) (:= i (+ 1 i)))
    ) )
```

Write down standard semantics for these new constructs. (Note: the FLFL code is only an example. The rest of this question concerns FLK!, not FLFL.) Do *not* present a desugaring into `callcc`, `label` or `goto`. You may use <u>`new`</u>, <u>`read`</u> and <u>`write`</u> if you wish. Make sure there is no unlimited growth of storage (unless the programmer allocates new cells, of course) even if the number of loop iterations is arbitrarily large.

The following Scheme code should help you answer the question.

```
1 ]=> (define a 3)
1 ]=> (define looper (lambda (body)
        (begin
          (callcc (lambda (c) (set! a c)))
          (body)
          (a 1)   )   )   )
1 ]=> (define mybody (lambda () (display "foo ")))
1 ]=> (looper mybody)
```

	6

Total: 40

YOU MAY USE THE SPACE BELOW FOR ROUGH WORK