**CSC173**
**Lambda Calculus 2013, some answers**

*Please write your name on the bluebook. You may use two sides of handwritten notes. There are 95 possible points, offering choice or insurance. Perfect score is 75. Stay cool and please write neatly.*

**1. Lambda Calculus Evaluation (10 min)**

Evaluate this $\lambda$ expression.

`((λp.λq.(p q) (λx.x λa.λb.a)) λk.k)`

Ans. Greg p. 276 (d)

**2. Function Equivalence (10 min)**

Show function (A) is equivalent to function resulting from expression (B) by applying each to (two) arbitrary arguments, which let's call `<fun>` and `<args>`. (Hint: here and throughout, be careful which name to expand: save work and reduce mistakes).

(A) `apply`

(B) `λx.λy.(((make-pair x) y) identity)`

Ans. Greg p. 276 (2.3 (b))

**3. Logical Operators (15 min)**

Recall we defined logical OR:
`def or = λx.  λy.(((cond true) y) x),`
which simplified to
`def or = λx.  λy.((x true) y).`

3.1 (5 min) : Prove that each of the 16 Boolean functions of two Boolean variables $x, y$ can be written as a C-style conditional in the form `a?b:c`, where
`a,b,c` are expressions that only use `x, y, true, false, not` (still not a minimal set). (Hint: a few English sentences is all you need: don't give 16 separate implementations!).

3.2 (5 min) : Write a *one-line, C-conditional-like (like OR above)* $\lambda$-calculus definition for logical EQUIVALENT (if and only if, $\Leftrightarrow$). Use only one `cond`, and you may also use `x, y` for the arguments, and `true, false` and `not`. Do not use `or, and,` or `implies`.

3.3 (5 min): Apply your $\lambda$-calculus definition from 3.2 to the two arguments `false` and `true` and show it evaluates to the correct answer. In your demonstration you can stay at the level of defined function names like `false, true, not...`, so that no $\lambda$s are needed. In fact, with judicious use of => ... => the demonstration can also be a one-liner. But if you want to work from more basic definitions, fine.

Ans: 3.1: the function is a 4-row, 3-col table with x and y values (T or F) in 1st two cols, f the function the 3rd col. wlog in a?b:c, let a = x. x value of T happens in 2 rows, and the b value

depends on y: it is TT or true, FF or false, or =y, or =not(y). Likewise the c value depends on y given not(x), similar argument. also, false = not(true) so don't need it. That's pretty incoherent (cb 2012). Another nice arg is to think of it as an if then else and note that you may not need the 2nd arg for some functions and if you do you can generate all dependencies you need.

Ans: Greg. p. 280 3.2

**4. Recursion (15 min)**

Recall from your exercises that this function `sum` finds the sum of numbers between `n` and `zero`:

```
def sum = recursive sum1 % apply recursive to sum1

def sum1 f n =
  if iszero n
  then zero
  else add n (f (pred n))}
```

4.1 (5 min). Similarly define functions `fac` and `fac1` that find the product of the numbers between $n$ and one (a.k.a. the factorial function). So `fac n` is equivalent to $n! = n*(n-1)*(n-2)*...*1$.

4.2 (10 min) Show the steps in evaluating
`fac three`.

Our goal here is to show how `recursive` works. We need to know its definition or its effect on its arguments: what does `(recursive f)` evaluate to? (Hint: $Y$'s mathematical definition.) We only want to see the names `fac1, recursive, mult, pred, three, two, one` in your evaluation (no $\lambda$s!), so use `=> ... =>` and `-> ... ->` to step past the conditional statements, and you never have to expand functions by `==`, either: keep application results at this high level. Use applicative-order evaluation where possible, and we'd expect about seven lines after the ones below, containing five more occurrences of `recursive`. Here we go....

```
fac three
recursive fac1 three
...
```

Ans: Greg 4.2 p. 284

**5. Different Integer Representation (25 min)** String theory was confirmed when a parallel, alternate world Threa was discovered occupying dimensions 5-8. Threans have a slightly different representation of integers in $\lambda$-calculus. They represent `zero` by `select-second`, not by `identity`. Sticking with the Earthian idea of formalizing integers as zero and recursive successors where $(0, 1, 2, ...) = (0, succ(0), succ(succ(0)), ...)$, they invented:

```
def zero == select-second == λs.λz.z
def one λs.λz.(s z)
def two λs.λz.(s (s z))
```
and so on.

5.1 (10 min.) They claim that this definition of `succ` works:

`def succ` $\lambda$`w.`$\lambda$`y.`$\lambda$`x.(y ((w y) x))`

Demonstrate that evaluating `(succ zero)` gives `one`. With that 'justification' let's assume the general result that `(succ (succ zero))` is `two`, etc.

5.2 (5 min.) **Addition:** Threans seem to use `succ` like an infix +, that is, `two succ three` is `five`, for instance. Demonstrate that in fact `((two succ) three)` evaluates to `(succ (succ three))`, which we believe is `five` by part 1 above. (Hint: rewrite `two` using its `def`, but you don't have to rewrite `succ` or `three`.)

5.3 (10 min.) **Multiplication!** Threans believe that the function

`def prod` $\lambda$`x.`$\lambda$`y.`$\lambda$`z.(x (y z))`

works to multiply `x` and `y` with `((prod x) y)`. Verify that `((prod two) two)` evaluates to `four`. (Hint: use $\alpha$-conversion and stay sane.)

## 6. (Fairly) Short Answers: (20 mins)

6.1 (5 min) State (explain briefly in plain English) the Church-Turing Hypothesis and the Church-Rosser Theorem (CRT).

6.2 (5 min) The Scheme syntax of the `let` function is:
`(let ([id val] ...)  exp1 exp2 ...)`.

Implement `let` with `lambda`: that is, write a Scheme expression that has the same effect as `let` but only uses the function `lambda`. (Hint: "only uses" means it does not use any other function!).

Ans:
```
(let ([id val] ...)  exp1 exp2 ...).
(( lambda (id ...)  exp1 exp2 ...)  val ...).
```

6.3 (10 min) Here's some Scheme:

```
(define call/cc call-with-current-continuation)

(define (compA cont-arg-A)
   (display cont-arg-A)
   (newline)
   (display "learn")
   (newline)
   (set! cont-arg-A (call/cc cont-arg-A))
   (display "heal")
   (newline)
   (set! cont-arg-A (call/cc cont-arg-A))
   (display "better")
```

```
    (newline)
   )

 (define (compB cont-arg-B)
    (display cont-arg-B)
    (newline)
    (display "discover")
     (newline)
     (set! cont-arg-B (call/cc cont-arg-B))
     (display "create")
     (newline)
    (set! cont-arg-B (call/cc cont-arg-B))
   (display "ever")
     (newline)
   )
```

(a) Why might I need that first `define`?

(b) Having `load`ed those functions, we type
`> (compa compb)`
at the Scheme Listener. What gets printed, if anything?

(c) Does `> (compb compa)` cause an error? Why or why not?

(d) Suppose we edit the file so that all `cont-arg-A`'s and `cont-arg-B`'s are changed to `foo`. Then we type
`> (compa compb)` at the Listener again. Does anything go wrong? What or why not?

Ans: 1.call/cc not defined in DrRacket!!

2.

```
> (compa compb)
#<procedure:compb>
learn
#<continuation>
discover
heal
create
better
```

3. no error, works fine:

```
> (compb compa)
#<procedure:compa>
discover
```

```
#<continuation>
learn
create
heal
ever
```

4. no problem, works fine. those args are local, can be called anything we please.