CPSC 311, 2011W1 – Midterm Exam #2 2011/11/09

Name: ___Sample Solution_____

Student ID: _____

Signature (required; indicates agreement with rules below):

_____

Q1:	20
Q2:	20
Q3:	20
Q4:	20
Q5:	20
	100

- You have 110 minutes to write the 5 problems on this exam. A total of 100 marks are available. Complete what you consider to be the easiest questions first!
- Ensure that you clearly indicate a legible answer for each question.
- If you write an answer anywhere other than its designated space, clearly note (1) in the designated space where the answer is and (2) in the answer which question it responds to.
- Keep your answers concise and clear. We will not mark later portions of excessively long responses. If you run long, feel free to clearly circle the part that is actually your answer.
- We have provided an appendix to the exam (based on your wiki notes), which you may take with you from the examination room.
- No other notes, aides, or electronic equipment are allowed.

  Good luck!

UNIVERSITY REGULATIONS

1. Each candidate must be prepared to produce, upon request, a UBCcard for identification.
2. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors or ambiguities in examination questions.
3. No candidate shall be permitted to enter the examination room after the expiration of one-half hour from the scheduled starting time, or to leave during the first half hour of the examination.
4. Candidates suspected of any of the following, or similar, dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action: (a) having at the place of writing any books, papers or memoranda, calculators, computers, sound or image players/recorders/transmitters (including telephones), or other memory aid devices, other than those authorized by the examiners; (b) speaking or communicating with other candidates; and (c) purposely exposing written papers to the view of other candidates or imaging devices.

   The plea of accident or forgetfulness shall not be received.
5. Candidates must not destroy or mutilate any examination material; must hand in all examination papers; and must not take any examination material from the examination room without permission of the invigilator.
6. Candidates must follow any additional examination rules or directions communicated by the instructor or invigilator

If you use this blank(ish) page for solutions, indicate what problem the solutions correspond to and refer to this page at the problem site.

Problem 1 [20%]

Vocabulary

1. Here's the Racket documentation for **`eval`**:

> The `eval` function takes a "quoted" expression or definition and evaluates it:
>
> ```
> > (eval '(+ 1 2))
> 3
> ```

Explain both: (1) why we might call **`eval`** "the perfect meta interpreter" and (2) why it is not a syntactic interpreter. **[Worth 6%]**

It's an interpreter in the sense that it takes a program as input and evaluates that program. It's a perfect meta-interpreter in that it directly implements the underlying language (Racket itself). Since it performs no explicit construction (on our part) of the language's features, it's *not* syntactic at all. (We don't even build any representation for values beyond Racket's!)

2. Write the letter of each description below in the blank next to the type of scope—static, dynamic, or "threaded" (the scope of state changes)—it ***best*** matches. All descriptions assume that a "narrower scope" overrides a "wider scope". A letter in the right blank is worth +0.5 marks; a letter in the wrong blank is worth -0.5; and you start with 0.5. **[Worth 4%]**

Static scope matches: E_G_

Dynamic scope matches: A_C_F

The "threaded scope" of state matches: B_D_

A. Until the currently evaluating function returns.

B. For the rest of the program.

C. In our FAE interpreter: in the current call to **`interp`**, any calls it makes to **`interp`**, calls they make to **`interp`**, and so on.

D. In a CPS program, until the continuation of the entire program is invoked.

E. In our FAE language, until the matching closing brace.

F. In a CPS function, until the continuation passed in is applied.

G. In the AST, at any node in the current subtree.

3. In Assignment #2, we pre-processed **`with`** expressions into function applications. A friend claims that in **`RCFWAE`** we can pre-process **`rec`** (the version where the named expression must be a syntactic function) into some combination of **`fun`** and **`app`**. Is this claim true? Briefly justify your answer. **[Worth 6%]**

Certainly we can. We can pre-process the rec expression into a simple with binding in which the named expression is made into a recursive function through use of the Y combinator.

4. For each part below, circle the best option: a NORMAL function allows us to do what's described on that line, a CONTINUATION allows us to do what's described, they BOTH do, or NEITHER does. (A "normal" function is a first-class function in a non-CPS program that uses no **`let/cc`**s or similar constructs. A "continuation" is a function provided by **`let/cc`** or the like.) **[Worth 4%]**

a. Which of these lets us re-enter a static context the program has already left behind?

NORMAL CONTINUATION ***BOTH*** NEITHER

b. Which of these lets us re-enter a dynamic context the program has already left behind?

NORMAL ***CONTINUATION*** BOTH NEITHER

c. Which of these lets us re-enter a state context (i.e., previous version of the store) the program has already left behind?

NORMAL CONTINUATION BOTH ***NEITHER***

Problem 2 [20%]

Surfacing Semantics

1. There is no semantic difference between two "copies" of the numerical value 3 in a BCFAE program. They may occupy different memory locations, but no BCFAE operation lets us discover that fact.

Let's define a copy of a box value to be another box value constructed with the same location, as with:

```
;; copy-box : boxV -> boxV
(define (copy-box box)
  (type-case BCFAE-value box
    [boxV (address) (boxV address)] ;; returns a copy, not box itself
    [else (error 'copy-box "Expected a box, received: ~a" box)]))
```

We'll extend the language so programmers can access this by writing **`{copy-box <BCFWAE>}`**.

Is there any semantic difference between a box and its copy? If so, illustrate with a brief program that behaves differently using **`copy-box`** (on an expression that produces a box) than without using **`copy-box`**. If not, explain why not. (Reminder: the specifically box-related operations are `setbox`, `openbox`, and `newbox`, plus now `copy-box`.) **[Worth 6%]**

No, there's no semantic difference. None of the non-box-related operations will tell us the difference. newbox is irrelevant (because we already have the boxes). setbox and openbox will treat both boxes identically, since they really operate on the memory location (which matches between the boxes). Fundamentally, the two problems are: (1) Values are immutable and constant. Only slots in the store can change. Two copies of an immutable object are indistinguishable from each other. (2) We have no direct way to access or test a value's memory location.

2. Give the outcome of the VCFAE program below with `fun` in the blank and with `refun` in the blank. (Reminder: `fun` uses call-by-value semantics; `refun` uses call-by-reference semantics.) **[Worth 8%]**

```
{with {y 1}
  {with {f {_____ {x} {seqn {set x {+ x 10}}
                            {seqn {set y {+ y 100}}
                              {+ x y}}}}}
    {f y}}}
```

Result with `fun`: 112

Result with `refun`: 222

Now, repeat the exercise, but with the expression **`{f y}`** on the last line replaced by **`{f {* y 10}}`**.

Result with `{f {* y 10}}` and `fun`: 121

Result with `{f {* y 10}}` and `refun`: error: {* y 10} is not an lvalue (i.e., has no mem location)

3. Consider the following program. For each of (1) eager evaluation, (2) lazy evaluation without caching, and (3) lazy evaluation with caching, indicate how many times the underlined expression is evaluated and what the program's result is. **[Worth 6%]**

```
{with {y {- 2 2}}
  {if0 y {+ y y}
         {foo bar}}}
```

EAGER – evaluated ___1_____ times; result: ___0_____

LAZY/no caching – evaluated ___3_____ times; result: ___0_____

LAZY/cached – evaluated ___1_____ times; result: ___0_____

Problem 3 [20%]

Tinkering with Innards

1. Add `alias?` to VCFAE. `{alias? <id> <id>}` returns **0** if the two identifiers are bound to the same store location and **1** otherwise. Fill in just the `alias?` case in `interp`. **[Worth 6%]**

```
(define-type VCFAE-value
  [numV (n number?)]
  [closureV (arg-name symbol?) (body VCFAE?) (env Env?)])

(define-type Env
  [mtEnv]
  [anEnv (id symbol?) (val number?) (rest-env Env?)])

(define-type Store
  [mtStore]
  [aStore (id number?) (val VCFAE-value?) (rest-store Store?)])

(define (lookup-env name env)
  (type-case Env env
    [mtEnv () (error 'lookup "free identifier ~a" name)]
    [anEnv (bound-name bound-value rest-env)
           (if (symbol=? bound-name name)
               bound-value
               (lookup-env name rest-env))]))

(define (lookup-store address store)
  (type-case Store store
    [mtStore () (error 'lookup "non-existent address ~a" address)]
    [aStore (bound-address bound-value rest-store)
            (if (= bound-address address)
                bound-value
                (lookup-store address rest-store))]))

(define (lookup name env store)
  (lookup-store (lookup-env name env) store))

(define (interp exp env store)
  (type-case VCFAE exp
    ...  ;; all other cases unchanged
    [alias? (id1 id2)
      (vxs (numV (if (= (lookup-env id1 env) (lookup-env id2 env))
                     0
                     1)) store)
      ;; or put vxs + numV inside, like (vxs (numV 0) store).
      ;; or very cool sol'n that definitely works, wrap
      ;; (interp (num ...) env store) around the if!

                                                              ]))
```

2. Consider an eager language with multiple-argument `fun`/`app` expressions pre-processed into nested single-argument `fun`/`app`s and `with`s pre-processed into function applications, as in Assignment #2. We modify it to expose the number of recursive calls "deep" we are in **`interp`** in the identifier **`deepness`**:

```
(define (interp expr old-env)
  (local ([define env (anEnv 'deepness
                              (numV (add1 (numV-n (lookup 'deepness old-env))))
                              old-env)])
    (type-case FAE expr
      …)))

(define (run sexp)
  (interp (pre-process (parse sexp)) (anEnv 'deepness (numV 0) (mtEnv))))
```

The simple program **`deepness`**, which previously caused an error, now evaluates to **`1`**.

(a) What does the following program evaluate to:
`{{fun {x y z} {+ {* x 100} {+ {* y 10} z}}} deepness deepness deepness}`
(No justification required, but **show your work** for partial credit.) **[Worth 4%]**

432

Why? Replacing the function with f, we have {f deepness deepness deepness}, which becomes {{{f deepness} deepness} deepness}. The rightmost deepness is at level 2 (not the root, but a child of the root). The middle is one level down, and the leftmost is two levels down. So, 2 + 3*10 + 4*100.

Note that since deepness alone evaluates to 1, then at the "root" level, we're already at 1, not 0.

(b) What does the program **`{with {f {fun {ignore} deepness}} {+ {f 0} 0}}`** evaluate to? (No justification required, but **show your work** for partial credit. Reminder: `with`s are pre-processed into function applications!) **[Worth 4%]**

`{with {f {fun {ignore} deepness}} {+ {f 0} 0}}`

becomes

`{{fun {f} {+ {f 0} 0}} {fun {ignore} deepness}}`

So, deepness is 3 levels deep in the tree (not the root, which is an app, not its child, which is a fun, but its child). Because of static scoping, the function will close over that environment. (In fact, it closes over the environment where deepness = 2 and then sets it to 3 when it interps its body.) This is just what we want to get the depth IN THE AST.

The result is {+ 3 0} = 3.

3. Convert the following functions into continuation-passing style (CPS). Assume that `empty?`, `and` (which is actually not a function), `first`, `rest`, and > are primitive functions that need not be converted. **[Worth 6%]**

(a)
```
(define (andall list)
    (if (empty? list)
      false
      (and (first list) (andall (rest list)))))
```

```
(define (andall/k list k)
    (if (empty? list)
      (k false)
      (andall/k (rest list) (lambda (it)
                              (k (and (first list) it))))))
```

(b)
```
(define (positive? n)
    (> n 0))
```

```
(define (positive?/k n k)
    (k (> n 0)))
```

Problem 4 [20%]

Other Languages are Programming Languages, Too!

1. Pipes in UNIX work somewhat like lazy evaluation, but processes in a pipe produce a few results immediately, deferring evaluation only if they get "too far ahead". In this problem, we explore this idea in our lazy interpreter (but we don't finish it!).

First, we add `cons_`, `first_`, and `rest_`, meaning the same as in Racket. We use `0` to mean `empty` and change `if0` so any value besides `0`, regardless of type, results in evaluating the else branch. So, we can test for an empty list with `{if0 list empty-result non-empty-result}`:

```
<CFWAE> ::= ...   ;; all the usual cases plus:
            | {cons_ <CFWAE> <CFWAE>}
            | {first_ <CFWAE>}
            | {rest_ <CFWAE>}

(define (strict val)
  (if (thunkV? val)
    (strict (interp (thunkV-body val) (thunkV-env val)))
    val))

(define-type CFWAE
  ...   ;; all variants look the same plus:
  [cons_ (lhs CFWAE?) (rhs CFWAE?)]
  [first_ (cexp CFWAE?)]
  [rest_ (cexp CFWAE?)])

(define-type CFWAE-Value   ;; Note the additional type for lists:
  [consV (lhs CFWAE-Value?) (rhs CFWAE-Value?)]
  [numV (n number?)]
  [thunkV (body CFWAE?) (env Env?)]
  [closureV (param symbol?)
            (body CFWAE?)
            (env Env?)])

;; parse does preprocessing, such as replacing with by function application.
(define (interp expr env)
  (type-case CFWAE expr
    ... ;;
    [if0 (c t e) (local ([define cresult (strict (interp c env))])
                   (if (and (numV? cresult) (= (numV-n cresult) 0))
                     (interp t env)
                     (interp e env)))]
    [cons_ (lhs rhs) (consV (thunkV lhs env) (thunkV rhs env))]
    [first_ (cexp)   (consV-lhs (strict (interp cexp env)))]
    [rest_ (cexp)    (consV-rhs (strict (interp cexp env)))]))
```

(a) **Give the result, as precisely as possible, of evaluating the following. [Worth 4%]**
`(interp (parse '{cons_ 1 0}))`

```
(consV (thunkV (num 1) (mtEnv)) (thunkV (num 0) (mtEnv)))
```

(b) Give the result of evaluating the following. **[Worth 3%]**

```
(strict (interp (parse '{with {f {fun {f} {cons {f f} {f f}}}}
                              {if0 {first {f f}} 0 1}})))
```

`(numV 1)` or, we would accept just 1, since we didn't ask for a precise result. Why? Because {first {f f}} evaluates to a consV, which is not (numV 0), so the if0 takes its else branch (under the new semantics described above).

(c) `constrict-n` simulates producing "a few results right away". It takes a number `n` and a `CFWAE-Value` and treats its `CFWAE-Value` argument like a binary tree, where internal nodes are `consV`s and leaves are any type of value besides `thunkV`s. It performs up to `n` calls to `strict` on the tree, working from left to right but does not evaluate any additional `thunkV`s. For example:

- `constrict-n` with an `n` of `0` does nothing to its argument.
- `(constrict-n 7 (thunkV (parse '{+ 1 2}) (mtEnv)))` evaluates to `(numV 3)`, as it would for any `n` ≥ `1`.
- ```
  (constrict-n 7
    (thunkV (parse '{cons_ 1 {cons_ {cons_ 2 {cons_ 3 4}}
                                    {cons_ 5 6}}}) (mtEnv)))
  ```
  calls `strict` on all the expressions on the same line as `parse` **except** the expression 4. It does **not** call `strict` on `4` or any expression on the following line.

Now, **complete the following definition of `constrict-n`. [Worth 7%]**

```
;; A value and a count of calls to strict made in this subtree.
(define-type VxC
  [vxc (value CFWAE-value?) (count number?)])

;; constrict-n : integer CFWAE-Value -> CFWAE-Value
(define (constrict-n n val)
  (local ([define (helper n val)
            (if (= n 0)
                (vxc val 0)
                (local ([define result (strict val)])
                  (type-case CFWAE-Value result
                    [consV (fst rst)
          (local ([define lresult (helper (- n 1) fst)]
                  [define rresult (helper (- n 1 (vxc-count lresult)) rst)])
             (vxc (consV (vxc-value lresult) (vxc-value rresult))
                  (+ 1 (vxc-count lresult) (vxc-count rresult))))
                                                                    ]
                    [else
                       (vxc result 1)
                                                                    ])))])
    (vxc-value (helper n val))))
```
```

2. The document typesetting language LaTeX uses "call-by-denotation" function application semantics. Consider the following code, which creates three new commands (functions) named `cbdSemanticsA`, `cbdSemanticsB`, and `cbdSemanticsC` and then creates a document that applies each command.

(Note: `\newcommand{\name[1]{body}}` defines a function named *name* with the body *body* and one argument referred to as `#1`. `\let\name=value` binds the identifier *name* to the value *value*. Anything in a function body that doesn't start with `\` or `#` is just plain text.)

```
\newcommand{\cbdSemA}[1]{ A arg: #1. }
\newcommand{\cbdSemB}[1]{ B arg unused. }
\newcommand{\cbdSemC}[1]{
  \let\foo=C
  C arg: #1.
}

\begin{document}
\cbdSemA{A}      % Applies cbdSemA to the text "A"            %% HERE!
\cbdSemB{\foo}   % Applies cbdSemB to the identifier foo
\cbdSemC{\foo}   % Applies cbdSemC to the identifier foo
\end{document}
```

This generates the text "`A arg: A.  B arg unused.  C arg: C.`" However, changing the line marked `HERE` to `\cbdSemanticsA{\foo}` would generate an error because `foo` is undefined.

Based on only these examples, **which of these best describes LaTeX's semantics? [Worth 6%]**

a. Static scope, eager evaluation

b. Static scope, lazy evaluation

c. Dynamic scope, eager evaluation

d. Dynamic scope, lazy evaluation

Briefly justify your answer.
(You shouldn't need all this space.)

In eager evaluation \foo would have failed for B if it would also have failed for A. \foo would be evaluated in the same way for both before the call. (In fact, we can tell simply because B and C's calls don't fail.)

In static scoping, C would still fail because \foo would remain unbound in the context in which it was possed to C. It can only give C as a result if the deferred expression \foo is drawing its value from its dynamic, rather than static, context.

Problem 5 [20%]

Extra Fun Problems!

1. Let's add encapsulation (simple "objects") to a version of RCFAE. The concrete syntax would be:

```
{with-object <id> {<RCFAE> …} <RCFAE>}
```

`with-object` takes a name (the `<id>`), a list of field expressions, and a body expression. It evaluates its body in an environment in which its name is bound to an "object". The object is a function that, given 0, gives back the first field in the list; given 1, it gives back the second field; and so on.

For example, the following should bind `evenodd` to an object whose 0 field is a function that tests whether a number is even (returning 0 if it is) and whose 1 field is a function that tests whether a number is odd. The program evaluates to 1 because 25 is **not** even. (Again, 0 means true in our languages!)

```
(run '{with-object evenodd {{fun {n} {if0 n 0 {{evenodd 1} {- n 1}}}}
                            {fun {n} {if0 n 1 {{evenodd 0} {- n 1}}}}}
                            {{evenodd 0} 25}}))
```

Complete the function below to pre-process away `with-object` in terms of other RCFAE expressions. Assume identifiers starting with * are reserved for the interpreter (so you can use them freely).

```
(define-type CFAE
  [num (n number?)]
  [id (name symbol?)]
  [add (lt CFAE?) (rt CFAE?)]
  [sub (lt CFAE?) (rt CFAE?)]
  [fun (arg-name symbol?) (body CFAE?)]
  [app (fun-exp CFAE?) (arg-exp CFAE?)]
  [if0 (test-exp CFAE?) (then-exp CFAE?) (else-exp CFAE?)]
  [rec (bound-id symbol?) (named-exp CFAE?) (body CFAE?)]
  [with-object (name symbol?) (fields (listof CFAE?)) (body CFAE?)])

(define (preprocess-with-object wo-exp)  ;; [Worth 6%]
  (type-case RCFAE wo-exp
    [with-object (name fields body)
      ;; Assumes we get to reserve IDs starting with * to ourselves.
      (local ([define (unroll fields n)
                (if (empty? fields)
                  (id '*with-object-field-out-of-bounds-error)
                  (if0 (sub (id '*select) (num n))
                    (first fields)
                    (unroll (rest fields) (add1 n))))])
        (rec name (fun '*select (unroll fields 0)) body))
]
    [else (error "Only works on with-objects!")]))
```

2. Consider the following function in CPS:

```
(define (c/k lof/k v k)
  (if (empty? lof/k)
      (k v)                                  ;; A (bold/italics)
      (c/k (rest lof/k) v                    ;; B (underlined, starts here,
        (lambda (r)                          ;;    spans 3 lines)
          ((first lof/k) r k))))))           ;; C (in bold/italics)
```

This code corresponds to a non-CPS function that used the call stack to "remember" pending computations. Three of its expressions are marked. A is `(k v)` on the third line of code. B spans the last three lines of code: `(c/k …)`. C is `((first lof/k) r k)` on the last line of code. All three expressions are tail calls in their immediately containing functions. However, each of A, B, and C corresponds to a simpler expression in the original code which may or may not have been in tail position.

(a) Give the name of every function called in **non-tail-position** in the code. **[Worth 2%]**

empty?, rest, and first. (Yes, first is called in non-tail-position.)

(b) Which of A, B, and C corresponds to a tail call in the original code, i.e., a function call in the original code where the call stack neither grew nor shrank? What indicates that it was a tail call in the original code? **[Worth 3%]**

C. We can tell because the continuation is passed along unchanged to whatever function (first lof/k) is.

(c) Which of A, B, and C corresponds to an expression in the original code that returned a value from the function, i.e., where the call stack *shrank*? What indicates that the call stack shrank? **[Worth 3%]**

A. Here, we actually apply the continuation, which corresponds to "returning" a value.

(d) How can it possibly be legal to apply `(first lof/k)` as if it were a function (on the last line)? **[Worth 2%]**

If lof/k is a list of functions (which it appears to be based on the somewhat informative name), then this would be perfectly legitimate. In general, a function call can certainly return a function value! That's true even if we only have higher-order functions, but we have first-class functions!

Note that lof/k is NOT a list of continuations. A continuation (in Racket) takes a value as its single parameter. This takes a value and a continuation.

3. Look back at Problem 3.1. Write an example of a VCFAE function that includes an `alias?` expression that evaluates to 0. Give the result of evaluating your program. **[Worth 4%]**

The fun/refun problem gives us a great example we can gut and repair:

```
{with {y 1}
  {with {f {refun {x} {alias? x y}}}
    {f y}}}
```

This program evaluates to 0.

And, as many students pointed out, this works nicely too:

{{fun {x} {alias? x x}} 0}

(Your example shouldn't take much space. We just ran out of exam.)

If you use this blank(ish) page for solutions, indicate what problem the solutions correspond to and refer to this page at the problem site.

If you use this blank(ish) page for solutions, indicate what problem the solutions correspond to and refer to this page at the problem site.

If you use this blank(ish) page for solutions, indicate what problem the solutions correspond to and refer to this page at the problem site.