CPSC 311, 2010W1 – Midterm Exam #2

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. List two important differences between the environment and the store in our interpreters. (***Unimportant*** differences would include, for example, "we call one `env` and the other `store`".)
[Worth 5%]

Two important differences: (1) The store threads through program execution (values have "dynamic extent"), while the environment's structure reflects the static structure of the program (identifiers have "static scope"). (2) The environment maps identifiers to either locations (or values if there is no store) while the store maps locations to values.

Many other answers exist such as, e.g., that "keys" in the store once "shadowed" will never be accessed again, while "keys" in the environment that are shadowed will come back into scope as program evaluation proceeds (barring unusual control flow!).

2. In evaluating a function call like `{f x}` in call-by-value semantics, we call `interp` on the expression `x`. However, calling `interp` on `x` under call-by-reference semantics is not helpful. Why not?
[Worth 5%]

`interp` evaluates expressions, returning their values, but `x`'s value is not what we need. We need the location associated with `x` so that we can bind `f`'s parameter to that location. (That binding ensures that `x` and `f`'s formal parameter are just different names for the same memory location.)

3. Clearly and succinctly describe the continuation of the underlined expression in the following Racket program. (Assume Racket evaluates the function and arguments in a function application expression in left-to-right order.) **[Worth 5%]**

```
(+ (expensive-function-call 2) (+ 1 2) (/ 12 2))
```

Using the "escaper" `lambda` notation, and assuming that `(expensive-function-call 2)` evaluates to `efc2-result`:

```
(lambda^ (RESULT) (+ efc2-result RESULT (/ 12 2)))
```

(In other words, the expensive function call has already completed; what remains is to add its value to the result passed to the continuation and the result of evaluating `(/ 12 2)`.)

4. If `k` is bound to a continuation, and the expression `(k 5)` correctly applies the continuation to the value 5, why doesn't it make sense to ask what the expression `(k 5)` evaluates to? It may help to think about or refer to the following code. **[Worth 5%]**

```
(+ 1 (let/cc k
       (+ 2 (k 5))))
```

An expression that applies a continuation doesn't evaluate to anything; it never returns.

(If it did, the example expression above would evaluate to 1 plus 2 plus whatever `(k 5)` evaluates to, but `(k 5)` escapes the current context entirely. The program's value is therefore 6, and we never add 2 to anything.)

Problem 2 [20%]

Surfacing Semantics

1. Give the result and the final value of `y` in the store for the following two VCFAE programs. Note that the programs differ only in the underlined word. (Reminder: a `fun` uses call-by-value semantics, while a `refun` uses call-by-reference semantics.) **[Worth 6%]**

```
{with {f {refun {x} {seqn {set x {+ x 1}}
                          x}}}
  {with {y 1}
    {f y}}}
```

Result: 2 Final value of `y`: 2

```
{with {f {fun   {x} {seqn {set x {+ x 1}}
                          x}}}
  {with {y 1}
    {f y}}}
```

Result: 2 Final value of `y`: 1

2. Consider the following code (from class) to multiply the elements of a list:

```
(define (mul-all list)
  (cond [(empty? list) 1]
        [(= (first list) 0) 0]
        [else (* (first list)
                 (mul-all (rest list)))]))
```

We altered this code to use continuations:

```
(define (mul-all2 list)
  (let/cc k
    (local ([define (helper list)
              (cond [(empty? list) 1]
                    [(= (first list) 0) (k 0)]
                    [else (* (first list)
                             (helper (rest list)))])])
      (helper list))))
```

(a) Which of the following is true of `mul-all`? **[Worth 4%]**
CIRCLE ALL THAT APPLY

(i), (iii), (iv). (If the first elt is 0, it returns 0 without inspecting more. The recursive call is not a tail call, since there's still a multiplication to be performed. It performs one multiplication per element up to a 0 or the end of the list. It returns to its caller, like any normal function.)

(i) `mul-all` does not inspect elements of a list after a `0`.

(ii) `mul-all` is tail recursive. (That is, all recursive calls are tail calls.)

(iii) `mul-all` performs at least two multiplications given the input `'(1 2 0 4)`.

(iv) Once it calculates the product of the list, `mul-all` returns that value to its caller.

(b) Which of the following is true of `mul-all2`? **[Worth 4%]**
CIRCLE ALL THAT APPLY

(i), (iv). (If the first elt is 0, it applies the continuation to 0 without inspecting more. The recursive call is not a tail call, since there's still a multiplication to be performed. If `helper` runs into a 0, it returns 0 from the top-level call to `mul-all2` (jumping entirely out of the recursive `helper` calls). Regardless, `mul-all2` returns to its caller, like any normal function; only `helper`'s return is potentially "skipped" by application of a continuation.)

(i) Neither `mul-all2` nor its helper inspects elements of a list after a `0`.

(ii) `mul-all2`'s helper function is tail recursive. (That is, all recursive calls are tail calls.)

(iii) `mul-all2` and its helper perform at least two multiplications given the input `'(1 2 0 4)`.

(iv) Once it calculates the product of the list, `mul-all2` returns that value to its caller.

3. Racket uses continuations to implement `web-read` in its web server framework.

An alternate version of `web-read` might post a webpage and then "block"—wait to return until a response comes back. In that case, there would be no explicit need for continuations.

Now, consider the following code:

```
(define (start req)
  (local ([define limit-text (web-read "Length Tester"
                                       "What is the limit?"))]
          [define limit (string->number limit-text)]
          [define text (web-read "Length Tester"
                                 "What is the text?")]
          [define surplus (- (string-length text) limit)])
    (web-page
     (if (> surplus 0)
         (format "That's ~a character(s) over the limit!" surplus)
         (format "Your text fits within the limit.")))))
```

Both versions work fine if the user: (1) opens the Length Tester website, (2) enters "25" on the first page and submits, and (3) enters "Hello world!" on the second page and submits. (The result is a page indicating that the text fits within the limit.)

Give an example of a series of user actions that would work correctly with the continuation-based implementation of web-read but would likely fail with the blocking version. Indicate how continuations enable the continuation-based implementation to handle your example while the blocking version cannot. **[Worth 6%]**

Essentially any series of actions that clones a tab and attempts to use both clone and original or uses the back button and attempts to alter already submitted responses. For example:

(1) Open the Length Tester website

(2) Enter "25" on the first page and submit

(3) Press the back button

(4) Enter "2" on the first page and submit

(5) Enter "Hi!" on the second page and submit.

The continuation version stores the continuation of the first web-read and can re-invoke it any number of times, such as when we use back to submit a page twice. (At step 4, it simply reinvokes the continuation to resume computation at the definition of limit-text with a new value.)

The blocking version cannot resume an old computation in this way. (Once the blocking version returns a value from web-read, however, that point in the computation is over/lost, and going back to it would require **much** more complex web-aware code than what we have above. Likely, either the second submit would fail with an error message or it would use the limit 25 rather than 2. It's very *unlikely* that it would use the *second* limit, however!)

Problem 3 [20%]

Tinkering with Innards

1. Convert the following functions into continuation-passing style (CPS). Assume that `empty?`, `cons`, `first`, `rest`, +, <=, and > are primitive and need not be converted. **[Worth 9%]**

(a)
```
(define (all-pos list)
    (cond [(empty? list) true]
          [(<= (first list) 0) false]
          [else (all-pos (rest list))]))
```

```
(define (all-pos/k list k)   ;; name change isn't necessary
    (cond [(empty? list) (k true)]
          [(<= (first list) 0) (k false)]
          [else (all-pos/k (rest list) k)]))
                ;; tail calls are easy to change!
```

(b)
```
(define (range i n)
    (if (> i n)
        empty
        (cons i (range (+ i 1) n))))
```

```
(define (range/k i n k)
    (if (> i n)
        (k empty)
        (range/k (+ i 1) n
          (lambda (rresult) (k (cons i rresult))))))
```

(c) The following function calls the previous two functions. Whether or not you've completed the previous two parts, you may assume that correct CPS versions of the functions are available.

```
(define (test-all-pos)
  (if (all-pos (range -5 5))
      'fail
      'pass))


(define (test-all-pos/k k)
  (range/k -5 5
    (lambda (rresult)
      (all-pos/k rresult
        (lambda (apresult)
          (if apresult      ;; No parens; apresult isn't a function!
            (k 'fail)
            (k 'pass)))))))  ;; Could also apply k to the whole if.
```

2. A friend decides to put together a little interpreter in Racket to simulate Java arithmetic, saying it will be easy to do based on our original AE calculator. You open up Racket and type `(/ 5 2)` at the read-eval-print loop. Racket prints `2.5`.

Use this result to explain why the Java simulator will be much more difficult to build than AE was. (It may help to discuss meta-interpreters and syntactic interpreters.) **[Worth 4%]**

Java's arithmetic rules differ drastically from Racket's. AE's arithmetic was simple to implement because we simply relied on Racket's arithmetic operations (using "meta-interpreter" style). However, this new interpreter will need to simulate the vast set of behaviours specificied by Java (in "synactic" style).

(Note: don't be fooled by the "syntax" in "syntactic". The problem isn't that Java's and Racket's syntax for arithmetic differs. This is an inconvenience but not a major one (and entirely irrelevant in the interpreter). The problem is that the *semantics* for arithmetic differs. The meaning of "5 divided by 2" (and ***many many*** other expressions) differs between Racket and Java. Want to see how much work it would be? You can get a sense from Section 4.2 of the Java Language Specification: http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2 .)

3. Add a `seqnA` statement to VCFAE. `{seqnA <exp1> <exp2>}` evaluates the first expression, then evaluates the second expression, and then returns the value of the ***first expression***. (`seqn` would return the value of the second expression instead.)

You need fill in only the `seqnA` case in `interp`. **[Worth 7%]**

```
(define-type ValuexStore
  (vxs (value VCFAE-value?) (store Store?)))

(define (interp exp env store)
  (type-case VCFAE exp
             [num (n)    (vxs (numV n) store)]
              ...

             [seqnA (exp1 exp2)
               (local ([define exp1-result (interp exp1 env store)]
                       [define exp1-store (vxs-store exp1-result)]
                       [define exp2-result
                                (interp exp2 env exp1-store)])
                 (vxs (vxs-value exp1-result) (vxs-store exp2-result))

                ;; Crucially: interp exp1 first, then exp2;
                ;;            thread the store through exp1,
                ;;            then exp2, then back to the result; and
                ;;            yield exp1's value, not exp2's









              ]
             ))
```

Problem 4 [20%]

Other Languages are Programming Languages, Too!

1. C includes the `&` (address-of) and `*` (dereferencing) operators to allow programmers direct access to store locations. Let's create a new language PFAE (P for Pointers) with call-by-value semantics *only* that extends our language with mutable variables (VCFAE) to include these operators. The new concrete syntax is:

```
<PFAE> ::= <number>
         ...
         | {seqn <PFAE> <PFAE>}
         | {set <symbol> <PFAE>}
         | {addr-of <symbol>}
         | {val-at <PFAE>}
```

`{addr-of x}` returns the store location at which `x`'s value is stored (as a `numV`). `{val-at 1000}` returns the value (if any) bound to store location `1000`. Trying to find the address of an unbound variable or the value at an undefined store location should produce an error.

(a) **Fill in the `addr-of` and `val-at` cases in `interp` below. [Worth 11%]**

```
(define-type PFAE
  [num (n number?)]
  [id (name symbol?)]
  ...
  [addr-of (id symbol?)]
  [val-at (exp PFAE?)]

  ;; You'll need this blank for part (c).
  [alias (id symbol?) (addr-exp PFAE?) (body PFAE?)]


  )

(define-type Env
  [mtEnv]
  [anEnv (id symbol?) (location number?) (more Env?)])

(define-type Store
  [mtSto]
  [aSto (location number?) (value PFAE-value?) (more 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 index sto)
  (type-case Store sto
     [mtSto () (error 'lookup "location not in store")]
     [aSto (loc val more)
           (if (= loc index) val
               (lookup-store index more))]))

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

(define-type ValuexStore
  [vxs (value PFAE-value?) (store Store?)])

(define (interp exp env store)
  (type-case PFAE exp
     [num (n)     (vxs (numV n) store)]
     ...
     [addr-of (id)
       (vxs (numV (lookup-env id env)) store)

       ;; The id is bound to its location in the environment;
       ;; so, that's the whole value (though we need a store, too).




       ]

     [val-at (exp)
       (local ([define exp-result (interp exp env store)])
         (vxs (numV-n (lookup-store (vxs-value exp-result))
              (vxs-store exp-result)))







       ]

     [alias (id addr-exp body-exp)
       (local ([define addr-result (interp addr-exp env store)])
         (interp body-exp
                 (anEnv id (numV-n (vxs-value addr-result)) env)
                 (vxs-store addr-result)))



       ]
))
```

(b) C is able to implement call-by-reference semantics using the address-of and dereference operators. For example, the following code swaps the values in x and y:

```
void swap(int * i1, int * i2)
{
  int temp = *i1;
  *i1 = *i2;
  *i2 = temp;
}

…
  int x = 5;
  int y = 7;
  swap(&x, &y);
…
```

It is *impossible* in our language so far to implement `swap`. Why? (Hint: The expressions `*i1 = *i2` and `*i2 = temp` are the ones that cause problems. The BNF for the language contains provides the information needed for this question.) **[Worth 3%]**

The only expression we have to set a variable's value (`set`) accepts only an identifier as its lvalue (the "expression" whose value is being set). So, we can only set variable's values, not dereferenced pointers like `{val-at i1}`.

(c) Now add a new type of expression `alias` to the language:

```
<PFAE> ::= ...
         | {alias {<symbol> <PFAE>} <PFAE>}
```

Like `with`, `alias` introduces a new binding into the environment and then evaluates its body.

Unlike `with`, the identifier is bound to an *existing* store location rather than a new one. The store location used is the result of evaluating the first PFAE in `alias`'s concrete syntax above—the one in the "named expression" position of a `with`.

For example, we can now translate the following C function that increments its parameter:

```
void increment(int * x) {
  (*x) = (*x) + 1;
}
```

Into PFAE code:

```
{with {increment {fun {star-x}
        {alias {x star-x}
          {set x {+ x 1}}}}}
   ...}
```

Add `alias` to the abstract syntax datatype definition above and add and complete an `alias` case in `interp`. [Worth 6%]

Problem 5 [20%]

Extra Fun Problems!

1. In our extended interpreter, we converted function definitions with multiple parameters to nested definitions of functions each with a single parameter. We also converted applications of functions to multiple arguments to multiple applications to a single argument each.

(a) Illustrate the results of this conversion on the following program. (For clarity, write the new program with only single-argument functions and applications in *concrete syntax*.) **[Worth 2%]**

```
{with {g {fun {x y} {+ x x}}}
  {g 1}}
```

```
{with {g {fun {x} {fun {y} {+ x x}}}}
  {g 1}}
```

(b) What value does the program from part (a) evaluate to with eager evaluation semantics? You needn't use proper Racket syntax, but clearly, concisely, and completely describe the value. **[Worth 2%]**

A closure of one parameter (`y`) that, when applied to any argument, evaluates to `2` (because `x` is bound to `1` in its enclosed environment).

(c) The style of function that takes its first argument and returns a function waiting on additional arguments is called "Curried". Imagine we were writing a Racket function `curry2` that takes a plain two-argument Racket function and returns a Curried two-argument function. So, given the definitions:

```
(define (add x y) (+ x y)
(define curried-add (curry2 add))
```

`(curried-add 3)` evaluates to a function that takes an integer and returns that integer added to `3`.

Complete the test case below to test that applying `two-arg-test` to "hello" and "world" behaves the same as Currying it using `curry2` and applying the result appropriately to the same arguments.
[Worth 3%]

```
(define (two-arg-test x y) (format "x = ~a, y = ~b\n" x y))
(test (two-arg-test "hello" "world")
      (((curry2 two-arg-test) "hello") "world")_____________________ )
```

(Other than the call to `curry2` itself, this is the same syntax that would result from transforming a multi-argument application as in the previous midterm's version of part (a) above.)

2. In class, we implemented a VCFAE interpreter that supported call-by-reference but not call-by-value. The interpreter failed spectacularly because `with` expressions are pre-processed into function applications like this:

```
{with {<id> <named-exp>} <body-exp>} ⇨
  {{fun {<id>} <body-exp>} <named-exp>}
```

In this problem, we investigate this further.

(a) Why is call-by-reference an inappropriate target for this translation? **[Worth 2%]**

Call-by-reference would demand that the `<named-exp>` be an identifier, but `with` expressions should work with any expression as the `<named-exp>`.

(b) More specifically, we can guarantee that the ***first*** expression evaluated by `interp` that is a function application or `with` will fail. Why? **[Worth 2%]**

The first such expression that `interp` (attemps to) evaluate must be the first identifier binding in its scope. Therefore, the named expression will either not be an identifier (resulting in an error in the call-by-reference code that tries to grab that identifier expression's symbol) or will be an unbound identifier (resulting in an unbound identifier reference when the call-by-reference code tries to lookup that identifier expression's symbol in the environment).

3. We might ask what the continuation of each of the underlined expressions is in the following program:

```
(define (reverse list)
  (if (empty? list)
      empty
      (cons (reverse (rest list))
            (first list))))


(reverse '(8 4 2 1))
```

Explain why we can only uniquely identify one of these two continuations. **[Worth 3%]**

A continuation closes over its dynamic context. Therefore, in general, we cannot fully specify the continuation of a particular program point statically; it depends how we reached that point in the program!

The surprise, then, is that we can uniquely identify *either* of these expressions' continuations. However, *in this particular program*, we only ever evaluate the underlined expression `empty` once. So, it has a unique continuation. (We evaluate the other expression four times; so, it does not!)

4. Conversion to continuation-passing style makes the process of growing and shrinking the "call stack" explicit. Consider the following function in CPS:

```
(define (f/k g/k list k)
  (if (empty? list)
      (k empty)
      (g/k (first list)
           (lambda (x)
             (if x
                 (f/k g/k (rest list)
                      (lambda (xs)
                        (k (cons x xs))))
                 (f/k g/k (rest list) k))))))
```

(a) Circle a place in the function where we can see the call stack growing. What indicates that the stack is growing? **[Worth 2%]**

Each of the continuations that introduces a new lambda is a point where the call-stack grows. The code in the lambda *besides* any call to `k` is extra "work to be done" added to the remembered context.

(b) Circle a place in the function where we can see the call stack shrinking. What indicates that the stack is shrinking? **[Worth 2%]**

Each location where we apply the continuation passed in (`k`) represents the call stack shrinking. (In particular, the expression `(k empty)` represents the call stack shrinking without it growing first.) At each application of a continuation, we "strip off" a lambda, actually doing some of the deferred work.

(c) Circle a place where we can see in CPS what was a tail call in the original (pre-CPS-conversion) code. What indicates that it's a tail call? (Note: the tail-call should be to a function that has now itself been converted to CPS.) **[Worth 2%]**

All function calls in a CPS program (except those we identify as "primitve") are tail calls. However, the ones where we pass `k` along directly as the continuation are tail calls. We can tell because passing k to a function explicitly says to that function "there's no more work belonging to this function to remember when you finish; just move on with the same 'work left to do' that was originally supplied to this function". In this particular case, only the final call to `f/k` is a tail call in the pre-transformation code.

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.