CPSC 311, 2010W1 – Practice Midterm Exam #1

Name: _____

Student ID: _____

Signature (required; indicates agreement with rules below):

_____

Q1:	20
Q2:	20
Q3:	20
Q4:	20
	80

- You have 110 minutes to write the 4 problems on this exam. A total of 80 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.
- TO BE DISCUSSED: whether you get a wiki'd appendix or open notes to refer to.
- 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

Practice Exam Notes

This practice exam contains examples of four of the five types of problems that will be on the actual exam. In particular, we will ask you:

- A problem requiring you to apply your knowledge of programming languages vocabulary. Our intent is that these questions will not be too challenging if you have a firm grasp of the vocabulary.

- A problem exploring the impact of differing semantics on small pieces of code. We will neither ask you to read nor produce long pieces of code in these examples. Instead, the intent is to illustrate semantic differences using concise and focused examples.

- A problem demanding analysis and modification of interpreters and related code (e.g., BNF specifications, parsers, pre-processors, type checkers, AST specifications, etc.) with a focus on exploring and changing the interpreted language's semantics.

- A problem analyzing existing programming languages in light of the concepts you've learned in class, possibly including comparing the language to one of our interpreters or building an interpreter to model some aspect of the language.

The problems will generally be similar in format to the ones provided here. Some may be longer or shorter than shown here. Each of the four problems will be worth 20% of the exam mark, for a total of 80%.

The remaining 20% will come from a series of problems whose format is unspecified. You can expect some of these latter problems to be very challenging to solve correctly or, in some cases, even to obtain partial credit on.

CAUTION: We've written these problems referring to concepts that we should cover in the first ½ or so of the course. We may well not reach some of these before Midterm Exam #1! (E.g., we may not have discussed stores and mutation by then.)

Problem 1 [20%]

Vocabulary

1. With a well-built parser, which one can contain more types of errors: abstract syntax or concrete syntax? Why?

2. Explain why a function expression's value must "close over its environment" (i.e., use closures) to make static scoping possible in an environment-passing interpreter for a language with first-class functions.

3. In what sense is a typical "if" expression "lazy", even in a language with eager evaluation?

4. Why does the inclusion of `set` expressions and a store in a language's semantics lead naturally to explicitly ordering the evaluation of the sub-expressions in an `add` expression?

Problem 2 [20%]

Surfacing Semantics

1. Consider the following two code snippets. One behaves differently under static vs. dynamic scoping (with eager evaluation), while the other behaves differently under lazy vs. eager evaluation (with static scoping). For each one, identify which design axis is relevant and indicate the result of executing the code (i.e., its value or behaviour of it yields no value) under each of the relevant pair of semantics.

```
{with {f 1}
  {with {temp {f f}}
    {{fun {ignore} 0} temp}
  }
}
```

Design Axis: _____ vs. _____

Result under each:

```
{with {x 5}
  {with {f {fun {ignore} x}}
    {with {x 4} {f 0}}}}
```

Design Axis: _____ vs. _____

Result under each:

2. Give an example of a BCFAE program that evaluates differently depending on whether the arguments of a + operation are evaluated left-to-right, right-to-left, or using the same store with which the entire + expression was evaluated.

Also, using your example, briefly explain why the last semantics (both arguments use the same store) violates the basic idea of a store.

Reminder: BCFAE is our basic CFWAE extended with boxes to allow mutation. It uses the following concrete syntax in addition to CFWAE's syntax:

```
<BCFAE> ::= ...

          | {newbox <BCFAE>}

          | {setbox <BCFAE> <BCFAE>}

          | {openbox <BCFAE>}

          | {seqn <BCFAE> <BCFAE>}
```

Problem 3 [20%]

Tinkering with Innards

Consider the following fragment of a simple FAE-style interpreter:

```
(define (interp expr env)
  (type-case FAE expr
    ...
    [num (n) (numV n)]
    [id (v) (lookup v env)]
    [fun (bound-id bound-body)
         (closureV bound-id bound-body env)]
    ...
    ))
```

1. Does this interpreter use static or dynamic scoping? Circle the best answer and justify your response (i.e., explain why the answer is “definite” or why it's “probable” but not “definite” as appropriate).

(a) Definitely static because:

(b) Probably static because:

(c) Probably dynamic because:

(d) Definitely dynamic because:

2. The bound-body of the function is ***not*** immediately evaluated when evaluating a `fun` expression. Does this alone indicate that the language uses lazy evaluation? Circle the best answer.

(a) Yes. Only a language with lazy evaluation would leave the body expression unevaluated.

(b) Probably. Otherwise, there's no good reason to leave the body expression unevaluated.

(c) No. Both lazy and eager semantics would leave the body expression unevaluated.

3. We said this language is "FAE-like". Assuming the identifier `f` is bound to a function, can we tell whether the following function application would be legal syntax in this FAE-like language: `{f 5}`? If so, indicate what about the interpreter fragment indicates this fact. If not, explain why not.

4. Fill in the following additional cases of the `interp` function to implement the primitive operations `cons`, `first`, and `rest`, which behave like their Racket counterparts. However, ensure that the language uses lazy evaluation semantics, forcing evaluation in these three operators only where necessary. The relevant portions of the AST and value data-types have been provided below. You may assume that the function `strict` below has been correctly implemented. (Note that `cons` in Racket does not require that its second argument be a list, nor does `rest` in Racket demand that it evaluate to a list.) **PERFORM ERROR-CHECKING AS NEEDED!**

```
(define-type FAE-Value
   ...
   [exprV (body FAE?) (env Env?)]  ;; lazily delayed expression
   [consV (lhs FAE-Value?) (rhs FAE-Value?)]
   [emptyV]
   ...)

(define-type FAE
   ...
   [cons (lhs FAE?) (rhs FAE?)]
   [first (expr FAE?)]
   [rest (expr FAE?)]
   ...)

;; strict : CFWAE-Value -> CFWAE-Value
;; Consumes a CFWAE-Value and generates the corresponding fully
;; evaluated CFWAE-Value (which can never be an exprV).
(define (strict value)
  ...)
```

```
(define (interp expr env)
   ...
   [cons (lhs rhs)









     ]

   [first (sub-expr)









     ]

   [rest (sub-expr)









     ]
   ...)
```

Problem 4 [20%]

Other Languages are Programming Languages, Too!

1. In Java, methods are the equivalent of functions. Based on your knowledge of Java, what type of methods does Java have: first-class, higher-order, or first-order? ***Briefly*** explain why.

2. Partway into its evolution, Java introduced the notion of an "inner class" declared within the lexical scope of an outer class. One prevalent use for this mechanism is to create instances of anonymous inner classes to pass as callback objects. For example:

```
// Based on: http://download.oracle.com/javase/tutorial/
//           uiswing/events/generalrules.html
public class MyClass extends Applet {
    int fieldA = 0;
    ...
    void someMethod() {
        int fieldB = 0;             // LINE 1
        fieldA--;                   // LINE 2
        ...
        someObject.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                fieldA++;           // LINE 3
            }
          });
        ...
    }
}
```

Changing **LINE 3** to refer to `fieldB` bound on **LINE 1**, would produce ***illegal*** Java code. Based on this fact, argue that Java's inner classes are not implemented using closures and static scoping like FAE's.

3. **LINE 3** as it stands ***is*** legal in Java, even though `fieldA` is declared in an "enclosing static scope" of **LINE 3**. Just as Java converts code like **LINE 2**'s **`fieldA--`** into **`this.fieldA--`**, Java converts the `fieldA` reference on **LINE 3** so it refers to a hidden variable storing a reference to the outer class's `this`. Why doesn't this solution give access to `fieldB`?

4. It ***would*** be legal to refer to `fieldB` at **LINE 3** if `fieldB` had been declared `final`. Oracle's (Sun's) Java compiler implements this by copying the values of all `final` variables in the enclosing scope referred to within an inner class into special "hidden" fields within that class. Complete the following implementation of a similar strategy in FAE by (1) filling in the blank portions of `find-free-ids` and (2) altering the code in `interp` as needed to change closures so that they only include the required identifiers.

(You may find `append` and `remove*` handy. `(append list1 list2)` generates a new list containing the elements of `list1` followed by the elements of `list2`.
`(remove* (list 'x) lst)` removes every occurrence of the symbol `x` from the list `lst`.)

```
(define-type Env
  [mtEnv]
  [anEnv (name symbol?) (value CFWAE-Value?) (env Env?)])

;; lookup : symbol Env -> boolean-or-CFWAE-Value
;; Consumes a symbol and an environment in which to find that
;; symbol.  Produces either the symbol's bound value or
;; false if the symbol is not found.
(define (lookup id env)
  (type-case Env env
             [mtEnv () #f]
             [anEnv (name value env)
                    (if (symbol=? id name)
                        value
                        (lookup id env))]))

;; abridge-env :: Env (listof symbol) -> Env
;; Consumes an environment and a list of identifiers and generates an
;; environment containing only the currently applicable bindings for
;; those identifiers.  (Ignores duplicate ids and ids that do not
;; appear in the env.)
(define (abridge-env env ids)
  (foldr (lambda (id r)
           (anEnv id (lookup id env) r))
         (mtEnv)
         (filter (lambda (id) (lookup id env))
           (remove-duplicates ids))))
```

```
;; find-free-ids :: FAE Env -> (listof symbol)
;; Consumes an expression and generates a list of the identifiers that
;; appear free in the expression.  (Duplicates in the returned list
;; are OK.)
(define (find-free-ids expr)
  (type-case FAE expr
             ...
             [num (n)


               ]

             [add (lhs rhs)









               ]

             [id (name)









               ]

             [fun (param body)









               ]

             [app (f arg-expr)









               ])
```

```
(define (interp expr env)
  (type-case FAE expr

    ...

    ;; YOU SHOULD FIX THIS IF NEEDED!  (Error-checking not necessary.)
    [fun (param body-expr)


       (closureV param body-expr env)]



    ;; YOU SHOULD FIX THIS IF NEEDED!  (Error-checking not necessary.)
    [app (fun-expr arg-expr)


       (local ([the-closure (interp fun-expr env)]


              [the-arg (interp arg-expr env)])


         (interp (closureV-body the-closure)


                 (anEnv (closureV-param the-closure)


                        the-arg


                        (closureV-env the-closure))))]


    ...))
```