CPSC 311, 2010W1 – Midterm Exam #1

Name: ____

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. How could you associate a single abstract syntax with multiple different concrete syntaxes? **[Worth 5%]**

2. The textbook calls `strict` to implement strictness points, as in the code for `num+`, which implements addition for interpretation of `add` expressions:

```
;; num+ : CFAE/L-Value CFAE/L-Value → numV
(define (num+ n1 n2)
  (numV (+ (numV-n (strict n1)) (numV-n (strict n2)))))
```

`strict` may interpret an expression whose evaluation was deferred. `interp` requires an environment for interpretation. Why doesn't `strict` need to consume an environment as a parameter to complete deferred evaluation? **[Worth 5%]**

3. Why is it harder to write large systems with dynamic scoping than static scoping? **[Worth 5%]**

4. In the two parts of this question, we consider whether a *particular instance* of an identifier is free or bound with respect to the nested set of scopes that encloses it.

(a) If the instance is *free* with respect to the whole program scope, can it be *bound* with respect to one of the smaller scopes that contains it? If so, give a brief example of a FWAE program that causes this to happen. If not, explain why not. **[Worth 2.5%]**

(b) If the instance is *bound* with respect to the whole program scope, can it be *free* with respect to one of the smaller scopes that contains it? If so, give a brief example of a FWAE program that causes this to happen. If not, explain why not. **[Worth 2.5%]**

Problem 2 [20%]

Surfacing Semantics

1. Why would the following Haskell code be erroneous if Haskell used eager rather than lazy evaluation? **[Worth 4%]**

```
ones = 1 : ones
```

2. 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 expression `{+ 2 1}` is evaluated and what the program's result is. **[Worth 8%]**

```
{with {x {foo bar}}
  {with {y {+ 2 1}}
    {+ y {if0 y {+ x y}
                {+ y y}}}}}
```

EAGER – evaluated _________ times; result: ____________________________

LAZY/no caching – evaluated _________ times; result: ____________________________

LAZY/cached – evaluated _________ times; result: ____________________________

3. Consider the following incomplete program:

```
{with {triplicate
        {with {x 3}
          {fun {y} {* x y}}}}
    {with ____________________________________
      {triplicate 5}}}
```

(a) Give code below that, if placed in the blank, would cause the program to yield a value *other than* 15 under *static scoping*. Indicate the result the program would generate. **[Worth 4%]**

Result: _________

(b) Give code below that, if placed in the blank, would cause the program to yield the value 15 under *static scoping* but *not* under *dynamic scoping*. Indicate the result the program would generate under dynamic scoping. **[Worth 4%]**

Result: _________

Problem 3 [20%]

Tinkering with Innards

1. Consider the following fragment of a "smart" parser for `FWAE` (supporting the expressions `add`, `with`, `fun`, `app`, `num`, and `id`). Its goal is to disallow numeric values (`numVs`) in the function position of a function application:

```
(define (parse sexp)
  (cond
    ...
    [(list? sexp)
     (case (first sexp)
       ...
       ;; Assume only applications fall into the next case:
       [else
        (local ([define fun-expr (parse (first sexp))]
                [define arg-expr (parse (second sexp))])
          (type-case FWAE fun-expr
            [num (n)   (error "Number used in function position")]
            [add (l r) (error "Number used in function position")]
            [else (app fun-expr arg-expr)]))])
    ...
    ]))
```

(a) Give an example of a brief program this parser *rejects* that attempts to apply a numeric value in the function position of a function application: **[Worth 4%]**

(b) Give an example of a brief program this parser *accepts* that attempts to apply a numeric value in the function position of a function application: **[Worth 5%]**

2. In Racket, the + identifier (and `*`, `-`, `/`, and others) is not specially parsed; it is a normal identifier that is pre-bound to the built-in addition function. So, `(+ 1 2)` is a function application that looks up the value bound to + and applies it to the arguments `1` and `2`.

Below is a modified FAE interpreter. Its functions take two arguments, and its function applications apply to two arguments (to match +). The interpreter also has a so-far-unused value type `internalFunV`. **Modify the `parse` and `run` procedures so that + is treated like a normal identifier but is pre-bound to an `internalFunV` implementing addition.** For example:

`{+ 1 2}`	evaluates to	3
`{{fun {f x} {f 5 x}} + 2}`	evaluates to	7 (`f` is bound to + and then applied to `5` and `2`)
`{{fun {x +} +} 1 2}`	evaluates to	2 (+ is bound to `2` and its value is returned)

The code follows. (Portions that require no changes are marked `DO NOT ALTER`.) **[Worth 11%]**

```
(define-type FAE                                                   ;
  [num (n number?)]                                                ; DO
  [id (name symbol?)]                                              ; NOT
  [add (lt FWAE?) (rt FWAE?)]                                      ; ALTER
  [fun (param1 symbol?) (param2 symbol?) (body FWAE?)]             ;
  [app (fun-exp FWAE?) (arg-exp1 FWAE?) (arg-exp2 FWAE?)])         ;


(define (parse sexp)
  (cond [(number? sexp) (num sexp)]
        [(symbol? sexp) (id sexp)]
        [(list? sexp)
         (case (first sexp)
           [(+)
            (add (parse (second sexp))
                 (parse (third sexp)))]
           [(fun)
            (fun (first (second sexp))
                 (second (second sexp))
                 (parse (third sexp)))]
           [else
            (app (parse (first sexp))
                 (parse (second sexp))
                 (parse (third sexp)))])]))


(define-type Env                                                   ;
  [mtEnv]                                                          ; DO
  [anEnv (id symbol?)                                              ; NOT
         (val FAE-Value?)                                          ; ALTER
         (more-subs Env?)])                                        ;
```

```
(define-type FAE-Value                                        ;
  [numV (n number?)]                                          ;
  [closureV (param symbol?) (body FAE?) (env Env?)]           ;
  [internalFunV (impl procedure?)])  ;; NEW VALUE TYPE        ; DO
                                                              ; NOT
;; interp : FAE Env -> FAE-Value                              ; ALTER
(define (interp a-fae env)                                    ;
  ...)                                                        ;

(define (run exp)
  (interp (parse exp) (mtEnv)))
```

Problem 4 [20%]

Other Languages are Programming Languages, Too!

1. Java's `&&` ("`and`") operator is short-circuiting: `a && b` evaluates to true if and only if each of `a` and `b` evaluates to true, but `b` is never evaluated if `a` evaluates to false. Below is an attempt to implement short-circuiting `and` as a CFWAE function, using the multi-argument syntax from our extended interpreter. **Note:** we treat 0 as true and 1 as false.

```
{with {and {fun {lhs rhs}
                {if0 lhs
                     {if0 rhs 0 1}
                     1}}}

    {and ____________________________}
```

(a) Fill in the blank above with a brief piece of code that would result in an error if `and`'s second argument were evaluated, but does *not* produce an error because it evaluates only `and`'s first argument. **[Worth 3%]**

(b) A friend argues that this is a correct implementation of short-circuiting `and` in a language with eager evaluation semantics because `if` behaves lazily, even in an eager language. Explain why this claim is false. (You may refer to your code from the previous part if you wish.) **[Worth 3%]**

2. Until version 2.2, Python did not support nested static scoping and closures. Instead, Python had exactly two static scopes at any point during execution: a global scope and a scope local to the current block of code. When creating a nested function, Python programmers could only close the function over identifiers bound outside its scope by using default parameters. So, the following code:

```
def createAdder(x):
  def adder(y, incr=x):
    return incr + y
  return adder
```

uses a default value for the `incr` parameter to create a function that consumes one argument (`y`) and adds `x` to that argument, much like the FWAE code:

```
{with {createAdder {fun {x}
                     {with {adder {fun {y} {+ x y}}
                       adder}}
  ...}
```

(a) Based on the code above, in what scope is the expression on the right-hand-side of a default parameter definition's = sign evaluated in Python? **[Worth 3%]**

(b) Why would the following Python code generate an error in Python before version 2.2? **[Worth 3%]**

```
def createDoubleAdder(x):
  def createAdder(y):
    def adder(z, incr1=x, incr2=y):
      return incr1 + incr2 + z

  return createAdder
```

(c) Python version 3 introduced a `nonlocal` keyword that allows access (technically, assignment) to identifiers outside the local scope. Alter the following BNF, AST declaration, parser, and interpreter for WAE to add an `outer` expression that allows access not to the current static binding for an identifier but to the next binding outward for that identifier. So the following program evaluates to 2 rather than 3:

```
{with {x 1}
  {with {x 2}
    {with {dummy1 0}
      {with {x 3}
        {with {dummy2 0}
          {outer x}}}}}}
```

Notes: You need not perform error-checking. We have left space where changes are required, and it should be more space than you need. We left the parser out; you should assume it works correctly for your BNF and abstract syntax. **[Worth 8%, 2% per blank section]**

```
; WAE ::= <num>
;       | { + <WAE> <WAE> }
;       | { with { <id> <WAE> } <WAE> }
;       | <id>
```

```
(define-type WAE
  [num (n number?)]
  [add (lhs WAE?) (rhs WAE?)]
  [with (name symbol?) (named-expr WAE?) (body WAE?)]
  [id (name symbol?)]









  )

(define-type Env
  [mtSub]
  [aSub (name symbol?) (value WAE?) (renv Env?)])
```

```
;; Looks up the first binding of name in env.
(define (lookup name env)
  (type-case Env env
             [mtSub () (error "free identifier")]
             [aSub (id val rest-env)
                   (if (symbol=? id name)
                       val
                       (lookup name rest-env))]))

;; Looks up the SECOND binding of name in env, not the first.
(define (lookup-next name env)
```

```
  )

(define (interp expr env)
  (type-case WAE expr
             [num (n) (numV n)]
             [add (l r) (+ (num-n (interp l env))
                           (num-n (interp r env)))]
             [with (id named-expr body)
                   (local ([define val (interp named-expr env)])
                     (interp body (aSub id val env)))]
             [id (v) (lookup v ds)]
```

```
  ))
```

Problem 5 [20%, 5% each]

Extra Fun Problems!

1. Our mutation-based RCFAE with eager evaluation interprets a `rec` by creating a cyclical environment in two steps: (1) generate an initially incorrect environment and (2) correct it after the named expression has been evaluated.

If RCFAE instead used lazy evaluation semantics, it would delay evaluation of the named expression, anyway. Does such an interpreter also need mutation create a cyclical environment? Why or why not? **[Worth 3%]**

2. Problem 3, Part 2 discusses Python's `global` keyword. Surprisingly, Python implements `global` in its parser. The Python interpreter never actually sees `global` at all. Clearly sketch the design of a pre-processor for Problem 3.2(c)'s WAE that pre-processes away the `outer` keyword before interpretation. **Note**: Your pre-processor should not run the WAE program! **[Worth 5%]**

(Hint: De Bruijn can help!)

3. 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 3%]**

```
{with {f {fun {x y z} {+ x y}}}
  {f 1 2}}
```

(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 3%]**

4. Performing the deferred evaluation of an expression closure in a language with lazy evaluation semantics is called “thawing”.

Consider a FWAE-like language with eager evaluation semantics that allows functions of 0 or 1 argument. We can use 0-argument functions (thunks) as expression closures (as we did in the assignment).

(a) Can we write a function that thaws a thunk? If so, implement `thaw` in the blank in the following program so that the program evaluates to 5. If not, explain why not. **[Worth 3%]**

```
{with {thaw {fun {x} ________________________________________}}
  {with {sample-thunk {fun {} 5}}
    {thaw sample-thunk}}}
```

(b) Can we write a function that defers evaluation of an expression (“freezes” it)? If so, implement `freeze` in the blank in the following program so that the program evaluates to 5. If not, explain why not. **[Worth 3%]**

```
{with {freeze {fun {x} ______________________________________}}
  {with {y {freeze {/ 1 0}}}
    5}}
```

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.