CPSC 311, 2011W1 – Midterm Exam #1 2011/10/05

Name: SAMPLE SOLUTION

Student ID: 12345678

Signature (required; indicates agreement with rules below):

Sample Solution

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. Imagine we are designing a well-crafted programming language for broad use. We have already decided on the abstract syntax.

As we design the concrete syntax, which is a more important goal:
(1) for the concrete syntax to be usable for programmers writing code in the language or
(2) for the concrete syntax to be easy to parse into abstract syntax?
Briefly justify your answer. **[Worth 5%]**

It might be possible to justify (2), but (1) seems the stronger answer. Concrete syntax is the programmer's "interface" to the programming language. If it's inconvenient for the interpreter, that's no big deal; just parse it into the more convenient abstract syntax. A language designer building a production programming language should be willing to build a somewhat more intricate parser in order to improve the programmer's productivity, particularly since parsers are not especially complex programs.

2. Our environments are effectively linked lists of bindings. Bindings earlier in the list override bindings later in the list. What does "earlier in the list" correspond to in terms of the shape of the abstract syntax tree? **[Worth 5%]**

"Earlier in the list" corresponds to **LOWER** in the abstract syntax tree.

3. Briefly explain why there is no difference between the static and dynamic context of a step in the interpretation of a WAE program (i.e., a program without functions)? **[Worth 5%]**

Unless it's interrupted by a function call, interpretation recursively descends the abstract syntax tree. So, without functions, a given point in the abstract syntax tree can only be reached dynamically by one path, which is the same path leading statically from the root of the tree. (There's no “jumping around in the tree”.)

4. In general in a language with functions, can you look at a particular point in a program (i.e., location on a page of code) and tell what identifiers are in scope in a language with dynamic scoping? If so, how? If not, why not? **[Worth 5%]**

No, you cannot tell this in general. Consider a program like this one:

```
{with {f {fun {ignore} x}}
  {+ {with {x 5} {f 0}}
     {with {y 5} {f 0}}
```

Under dynamic scoping, the *second* call to the function bound to f generates an error. In other words, at different points in the execution of the program, the dynamic scope of the function includes different sets of identifiers.

Problem 2 [20%]

Surfacing Semantics

1. For each of the following brief pieces of CFWAE code, indicate whether it would be more efficient to evaluate it using eager or lazy evaluation (without caching) and why. **[Worth 6%]**

```
{with {x {+ 1 2}}
  {with {y {+ x x}}
    {+ y y}}}
```

More efficient under: (circle one) EAGER EVALUATION LAZY EVALUATION

Because:

This one will be more efficient under EAGER evaluation semantics since identifiers are used repeatedly.

```
{with {x {+ 1 2}}
  {with {y {- 5 5}}
    {if0 y 0 {/ {* x x} y}}}}
```

More efficient under: (circle one) EAGER EVALUATION LAZY EVALUATION

Because:

This one will be more efficient under LAZY evaluation semantics since x's value will never need to be calculated. Notice, however, that NEITHER semantics performs the expensive computation at the end.

2. Indicate the outcome of interpreting the CFWAE program below under static and under dynamic scoping. **[Worth 6%]**

```
{with {countdown {fun {n} {if0 n 0 {countdown {- n 1}}}}}
  {countdown 3}}
```

Outcome under static scoping:

Unbound identifier: countdown.

Outcome under dynamic scoping:

0

3. ML is a "pure functional programming language" similar in many ways to Haskell. For each of the following ML programs, indicate whether evaluating it would tell you whether ML uses eager or lazy evaluation and why. **[Worth 4%]**

(Note: `div` is integer division and causes an error on divide-by-zero. `fun` introduces a function definition, where the first identifier is the function name, remaining identifiers are parameters, and the body occurs after the = sign.)

```
if (1 > 0) then 1 else (1 div 0);
```

Would this indicate whether ML uses lazy evaluation? YES NO

Because:

No. Both an eager and a lazy language would produce 1.

```
fun my_if c t e = if c then t else e;
my_if (1 > 0) 1 (1 div 0);
```

Would this indicate whether ML uses lazy evaluation? YES NO

Because:

Yes. In an eager language, the division by zero error will occur before the function is called. In a lazy language, it will never occur (since the condition is true).

4. In Haskell, `[x..y]` where `x` and `y` are integers and `y ≥ x` is the list `[x, x+1, x+2, …, y]`.

Now, consider the following code:

```
allones 0 = True
allones n = if n `mod` 10 == 1
            then allones (n `div` 10)
            else False

result = filter allones (map (\x -> x * x - 1) [1..10000000])
```

The code uses the "pipeline" style common in lazy languages. What is the advantage of lazy evaluation for the pipelined style? **[Worth 4%]**

The pipeline is able to work on only a small "active" portion of the list at a time. In an eager language, the whole "work queue" must be instantiated at each step. In this case, that would consume a great deal of memory. That's particularly unfortunate here when it's likely few of the results match what's desired.

(This is especially obvious when an independent process that runs for an unknown duration is generating the input to the pipeline. In that case, it's quite nice not to have to wait for that process to finish before proceeding with the remainder of the pipeline.)

Problem 3 [20%]

Tinkering with Innards

1. We wish to add a `read-eval` construct to a parser/interpreter for the FWAE language. To the programmer, use of `read-eval` looks like a 0-argument function application. When interpreted, it reads concrete syntax from standard input and evaluates it in the current environment.

For example, when interpreted, `{with {x {read-eval}} {+ {read-eval} 0}}` waits for input from the user twice. If the user types `2` and then `{+ x 3}`, the result of the program will be `5`.

(a) Add `read-eval` to the EBNF and abstract syntax definition of the CFWAE language below. **[Worth 4%]**

```
<CFWAE> ::= <num>
     | {+ <CFWAE> <CFWAE>}
     | <id>
     | {if0 <CFWAE> <CFWAE> <CFWAE>}
     | {with {<id> <CFWAE>} <CFWAE>}
     | {fun {<id>} <CFWAE>}
     | {<CFWAE> <CFWAE>}
     | {read-eval}
```

```
(define-type CFWAE
  [num (n number?)]
  [add (lhs CFWAE?) (rhs CFWAE?)]
  [id (name symbol?)]
  [if0 (c CFWAE?) (t CFWAE?) (e CFWAE?)]
  [fun (param symbol?) (body CFWAE?)]
  [app (fun-expr CFWAE?) (arg-expr CFWAE?)]
  [read-eval]



)
```

(b) Assuming that the remainder of the parser is implemented correctly, complete the following parser for FWAE with the `read-eval` construct. **[Worth 2%]**

```
(define (parse sexp)
  (cond
    …                           ;; Assume remaining cases are correct
    [(list? sexp)
     (case (first sexp)
       ['+   (add (parse (second sexp)) (parse (third sexp)))]

       ;; Implement the read-eval case.
       ['read-eval (read-eval)]









       …)]))                    ;; Assume remaining cases are correct
```

(c) Assuming that the parser and the remainder of the interpreter work correctly, complete the `read-eval` case for the interpreter below. Note: in Racket evaluating `(read)` reads input from the user and returns the result as an s-expression. **[Worth 4%]**

```
(define (interp-env env ast)
  (type-case FWAE ast
    [add (lhs rhs) (numV (+ (numV-n (interp env lhs))
                            (numV-n (interp env rhs))))]

    ;; Implement the read-eval case.

    [read-eval () (interp-env env (parse (read)))]









    …))                             ;; Assume remaining cases are correct
```

2. Consider the following modification to an otherwise working FWAE implementation:

```
;; appends the second Env to the first to produce a new Env.
(define (append-env env1 env2)
  (type-case Env env1
    [mtEnv () env2]
    [anEnv (name value restEnv)
           (anEnv name value (append-env restEnv env2))]))

(define (interp-env env ast)
  (type-case FWAE ast
    …       ;; assume all other cases work correctly and normally
   [app (fun-expr arg-expr)
         (local ([define value (interp-env env arg-expr)]
                 [define fun-value (interp-env env fun-expr)]
                 [define fun-param (closureV-param-name fun-value)]
                 [define fun-body (closureV-body fun-value)]
                 [define fun-env (closureV-env fun-value)])
           (interp-env (append-env env fun-env) fun-body))]))
```

(a) We said that a function's job is to: (1) defer evaluation of the function body, (2) close over the static context of the function's definition, (3) bind the function's parameter to a value from the dynamic context of the function's application, and (4) otherwise be opaque to the dynamic context of the function's application.

The original code achieved all of these goals. Which of these goals does the new version of the code achieve? **[Worth 4%]**

Deferring evaluation of the body and (sort of) closing over the static context. HOWEVER, the dynamic context can override the static context.

Not binding the parameter, nor being opaque to the dynamic context at all.

(b) What is the outcome of interpreting the following code with this interpreter? **[Worth 3%]**

```
{with {f {with {x 10}
           {fun {ignore} x}}}
  {with {x 100} {f 1}}}
```

Outcome is: 100

(c) What is the outcome of interpreting the following code with this interpreter? **[Worth 3%]**

```
{with {f {with {x 10}
           {fun {x} {+ x y}}}}
  {with {y 100} {f 1}}}
```

Outcome is: 110

Problem 4 [20%]

Other Languages are Programming Languages, Too!

Go (Google's new systems programming language) and Python both provide lightweight syntax for functions to return multiple results. In Go, we can define a division function that returns a result and an error flag that is true if the denominator is 0:

```
func carefuldiv(num int, den int) (int, bool) {
  if den == 0 {
    return 0, true
  }
  return (num / den), false
}
```

We can call `carefuldiv` and separate out its results:

```
  quo, succeeded := carefuldiv(1, 0)
```

`succeeded` will be false, since we divided by zero. However, the following will not compile in Go:

```
      combined_results := carefuldiv(1, 0)
```

The equivalent code *does* work in Python. So, Go includes syntax for functions to hand multiple values to assignment statements. Python introduces a new type of value that wraps up many individual values and is usable in any context where arbitrary values are legal.

(a) To try Go's solution, we allow multiple bodies (return values) in a function, introduce the `bind-multi` expression to use multiple return values, and change `closureVs` to allow multiple bodies:

```
FWAE ::= ...            ;; all others cases unchanged
     | { fun { <id> } <FWAE> ... }
     | { bind-multi { <id> ... } {<FWAE> <FWAE>} <FWAE> }

(define-type FWAE
  ...                   ;; all others variants unchanged
  [fun (p-name symbol?) (bodies (listof FWAE?))]
  [bind-multi (ids (listof symbol?)) (fun-expr FWAE?)
              (arg-expr FWAE?) (body FWAE?)])

(define-type FWAE-Value
  [numV (n number?)]
  [closureV (p-name symbol?) (env Env?) (bodies (listof FWAE?))])
```

A `bind-multi` evaluates its `fun-expr`, evaluates its `arg-expr`, and applies the function value to the argument value, evaluating each of the function bodies and binding them to the corresponding id. In the resulting environment, it evaluates the body. For example:

```
 {bind-multi {x y} {{fun {b} {* 2 b} {* 3 b}} 1} {- x y}}
```

Evaluates to `-1`, since `x` is bound to `2` and `y` to `3`.

Finally, recall our Env definition:

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

Complete the implementation of the interpreter below. Feel free to use `interp-env*`.

```
;; bind-all : (listof symbol) (listof WAE-Value) Env -> Env
;; Extends the Env by binding the symbols to the corresponding values.
;; Give an error if the two lists are not the same length.
(define (bind-all ids vals env)
  ;; [Worth 3%]
  (cond [(and (empty? ids) (empty? vals)) env]
        [(or (empty? ids) (empty? vals)) (error "Error!!")]
        [else (anEnv (first ids) (first vals)
                     (bind-all (rest ids) (rest vals) env))])

  )


(define (interp-env* env aes)
  (if (empty? aes)
    empty
    (cons (interp-env env (first aes)) (interp-env* env (rest aes)))))

(define (interp-env env an-ae)
  (type-case FWAE an-ae
    ...                 ;; all other cases unchanged
    ;; fun and app cases changed slightly; app case left out for space
    [fun (p-name bodies) (closureV p-name env bodies)]
    [app (fun-expr arg-expr) ...left out...]

    [bind-multi (ids fun-expr arg-expr body)
      ;; [Worth 5%]
     (local ([define fun-value (interp-env env fun-expr)]
       [define arg-value (interp-env env arg-expr)]
       [define results (interp-env* (anEnv (closureV-p-name fun-value)
                                           arg-value
                                           (closureV-env fun-value))
                                    (closureV-bodies fun-value))])
       (interp-env (bind-all ids results env) body))

    ]))
```

(c) To try Python's solution, we introduce a new `tupleV` value type, a new `tuple` expression that creates a `tupleV`, and the `bind-multi` expression that binds the parts of a tuple to multiple ids:

```
FWAE ::= ...              ;; all others cases unchanged
      | {tuple <FWAE> ... }
      | { bind-multi { <id> ... } <FWAE> <FWAE> }

(define-type FWAE
  ...                     ;; all others variants unchanged
  [tuple (exprs (listof FWAE?))]
  [bind-multi (ids (listof symbol?)) (named-expr FWAE?) (body FWAE?)])

(define-type FWAE-Value
  [tupleV (values (listof FWAE-Value?))]
  [numV (n number?)]
  [closureV (param-name symbol?) (env Env?) (body FWAE?)])
```

A `bind-multi` evaluates its `named-expr`, which must yield a tuple value. It binds each value in the tuple to the corresponding id. In the resulting environment, it evaluates the body. For example:

```
 {bind-multi {x y} {tuple 2 3} {- x y}}
```

Evaluates to `-1`, since `x` is bound to `2` and `y` to `3`.

Complete the implementation of the interpreter below, assuming that the `bind-all` function from the previous part is also available here.

```
(define (interp-env env an-ae
  (type-case FWAE an-ae
    ...                   ;; all other cases unchanged
    [tuple (exprs)
      ;; [Worth 2%]
      (tupleV (interp-env* env exprs))



      ]
    [bind-multi (ids named-expr body)
      ;; [Worth 4%]
      (interp-env (bind-all ids
                     (tupleV-values (interp-env env named-expr)) env)
         body)




      ]))
```

(d) Is it possible to create a collection of multiple values that itself contains a collection of multiple values in the Go-like version? If so, give a short code example showing how you would do it. If not, explain why not. **[Worth 3%]**

No. A function that returns multiple values can only be used as the binding value producer in a bind-multi. Therefore, immediately after being "constructed", it is "torn down". There's never an opportunity to nest.

(And, it appears there is no way to nest tuples in Go, either.)

(e) Is it possible to create a collection of multiple values that itself contains a collection of multiple values in the Python-like version? If so, give a short code example showing how you would do it. If not, explain why not. **[Worth 3%]**

Yes. For example:

```
{tuple {tuple 1 2} 3}
```

Problem 5 [20%]

Extra Fun Problems!

1. In Assignment #2, we converted functions like {fun {x y} <body>} into {fun {x} {fun {y} <body>}}. Similarly, we converted applications like {f x y} into {{f x} y}.

Assume this is our code for interpreting the `app` case:

```
[app (fun-expr arg-expr)
     (local ([define value     (interp-env env arg-expr)]
             [define fun-value (interp-env env fun-expr)]
             [define fun-param (closureV-param-name fun-value)]
             [define fun-body  (closureV-body fun-value)]
             [define fun-env   (closureV-env fun-value)])
       (interp-env (anEnv fun-param value fun-env) fun-body))]
```

And, consider the following code:

```
{with {f {fun {x y z} {+ {+ x y} z}}}
  {f a {fun {w} w} {/ 1 0}}}
```

(a) Assume the interpreter rewrites multi-argument functions into nested single-argument functions. Rewrite `{f a {fun {w} w} {/ 1 0}}` by hand so that it uses only one-argument function applications. Write your answer in **concrete** (not abstract) syntax. **[Worth 2%]**

```
{{{f a} {fun {w} w}} {/ 1 0}}}
```

(b) Our interpreter would signal an error on this code, but only one. What are **all** the errors in the code? **[Worth 2%]**

Unbound identifier a. Function value used where numerical value was expected (in +). Division by zero.

(c) Based on the given `app` case from `interp`, which error would you hypothesize will be reported? State any assumptions you make about the Racket interpreter to come to this conclusion. **[Worth 3%]**

The most likely hypothesis seems to be that the division by zero will be generated, because the argument expression is interpreted first in the `app` case. This assumes Racket evaluates `define` bodies in order.

(d) When the interpreter evaluates an expression like `{<fun> <arg1> <arg2> <arg3>}`, what is the relationship among the environments in which the expressions `<fun>`, `<arg1>`, `<arg2>`, and `<arg3>` are evaluated? **[Worth 3%]**

The same environment as the one the entire expression is evaluated in.

2. Do you think the following definition would be legal in Haskell? If so, why do you think it would work, and what type would Haskell assign to `recursion`? If not, what will happen and why?

```
recursion = recursion
```

It may help to remember what happens with lazy evaluation in our interpreter when we evaluate `{with {x y} 5}`. **[Worth 4%]**

Yes, it will work. Just as the y in our program above is never "touched", Haskell will never "touch" the recursion identifier (which would lead to infinite recursion). Since there's no information whatsoever about its type, and Haskell infers the most general type, it should be "a", that is, an entirely unbound type variable.

3. A possible "optimization" to closures is to remove from the enclosed environment all bindings for identifiers that are not free within the function body, but this optimization is not semantics-preserving in the presence of problem 3.1's `read-eval` expression. Write a small program using `read-eval` and provide appropriate user input that would evaluate incorrectly under this optimization. **[Worth 6%]**

Program:

```
{{with {x 5} {fun {y} {read-eval}} 0}
```

User input:

```
x
```

Intended result: __5___ **"Optimized" result:** Unbound id x

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.