Comp 311: Sample Midterm Examination

October 29, 2007

Name: _____

Id #: _____

Instructions

1. The examination is closed book. If you forget the name for a Scheme operation, make up a name for it and write a *brief* explanation in the margin.
2. Fill in the information above and the pledge below.
3. There are 6 problems on the exam, totaling 100 points on the exam.
4. You have three hours to complete the exam.

Pledge:

Problem 1. (10 points) Al Gaulle, a programmer for Kludge, Inc., is designing a simple extension language for a business software package. He is proposing the grammar for imperative Jam (Project 4) except for the following revision to the syntax for `if`-expressions:

```
<if-exp> ::= if <exp> then <exp> else <exp>
           |  if <exp> then <exp>
```

Do you see any problems with this specification (besides the questionable use of Jam as the foundation for his language), particularly his revision to Jam syntax? State your criticism precisely.

Problem 2. (10 points)

Al Gaulle is responsible for maintaining a Scheme program written by another programmer. In the middle of the program, Al notices the application

```
((lambda (f) (lambda (x) (map f x)))
   (lambda (z) (+ x z)))
```

Al decides to optimize the program by reducing the application (using substitution) to

```
(lambda (x) (map (lambda (z) (+ x z)) x)
```

Did he optimize the program correctly? Why or why not?

Problem 3. (20 points) Recall that Scheme `let` construct (which is *not* recursive) expands into `lambda` expressions as follows:

```
(let [(x1 E1)
      (x2 E2)
      ...
      (xn En)]
  E)}
```

abbreviates

```
((lambda (x1 x2 ... xn) E) E1 E2 ... En)
```

Similarly, the `let*` construct expands into `let` expressions as follows:

```
(let* [(x1 E1)
       (x2 E2)
       ...
       (xn En)]
  E)
```

abbreviates

```
(let [(x1 E1)]
  (let [(x2 E2)]
    ...
      (let [(xn En)]
        E)...))
```

The other binding form in the Scheme `let` family is `letrec`; it has the same scoping rules as the Jam recursive let.

For each of the two expressions on the next page, circle each binding occurrence of a variable and draw arrows from each bound occurrence back to the corresponding binding occurence. For example, given the expression

```
(lambda (x) (+ x 1))
```

the correct answer is:

```
(lambda (x) (+ x 1))
```

1.
```
(let*
    [(fib (lambda (n)
            (letrec
              [(fibhelp (lambda (m fn-1 fn-2)
                          (let [(fn (+ fn-1 fn-2))]
                            (if (zero? m)
                                fn
                                (fibhelp (sub1 m) fn fn-1)))))]
              (if (< n 2)
                  1
                  (fibhelp (sub1 n) 1 1)))))
     (fib100 (fib 100))]
    (* fib100 fib100))
```

2.
```
(let* [(pair (lambda (x y)
               (let [(x x)
                     (y y)]
                 (lambda (msg)
                   (cond
                     [(eq? msg 'first) x]
                     [(eq? msg 'second) y]
                     [else (error 'pair "illegal method name ~a" msg)])))))
       (pair (pair 1 2))]
  (pair 'first))
```

Problem 4. (20 points) The following `Eval` function interprets a subset of Jam with call-by-value semantics; it is a pruned version of the solution to Assignment 2. For the sake of pedagogic simplicity, functions have been restricted to a single argument, some Jam constructs and primitives have been eliminated, and almost all error checking has been removed.

Modify `Eval` (which appears on the next page) to support a recursive `let` construct that binds only one identifier

```
let fact := map n to ...;
in fact(100)
```

The abstract syntax for the construct is

```
(define-struct let-exp (id rhs body))
```

where `id` is the new identifier, `rhs` is an expression specifying what `id` is bound to, and `body` is the expression to be evaluated in the extended environment. A space gap has been left in the body of `Eval` so that you can easily insert your modification.

```
(define-struct num-exp (arg))
(define-struct id-exp (arg))
(define-struct biop-exp (rator rand1 rand2))
(define-struct map-exp (var body))          ; unary functions only
(define-struct app-exp (rator rand))        ; unary functions only
(define-struct binding (var val))
(define-struct closure (parm body env))     ; unary functions only
```

```
(define-struct let-exp (id rhs body))

(define Eval
  (lambda (exp env)
    (cond  ; bool, prim, unop, and if eliminated for pedagogic simplicity
      [(num-exp? exp) (num-exp-arg exp)]
      [(id-exp? exp) (binding-val (id-lookup (id-exp-arg exp) env))]
      [(biop-exp? exp)
       (local [(define rator (biop-exp-rator exp))
               (define rand1 (Eval (biop-exp-rand1 exp) env))
               (define rand2 (Eval (biop-exp-rand2 exp) env))]
         (cond [(eq? rator '+) (+ rand1 rand2)]
               [(eq? rator '-) (- rand1 rand2)]
               [else
                 (error 'Eval "unrecognized binary operator ~a" rator)]))]
      [(map-exp? exp)
       (make-closure (map-exp-var exp) (map-exp-body exp) env)]
      [(app-exp? exp)
       (Apply (Eval (app-exp-rator exp) env) (Eval (app-exp-rand exp) env))]
      [else (error 'Eval "illegal expression: ~a" exp)])))

(define extend
  (lambda (env var val)
    (cons (make-binding var val) env)))

(define id-lookup ; returns binding pair not binding-val
  (lambda (id env)
    (cond
      [(eq? id (binding-var (car env))) (car env)]
      [else (id-lookup id (cdr env))])))

(define Apply
  (lambda (head arg-val) ; head must be a closure
    (local
```

```
  [(define parm (closure-parm head))
   (define body (closure-body head))
   (define env (closure-env head))]
(Eval body (extend env parm arg-val)))))
```

Problem 5. (20 points) Let Jam have the semantics specified in assignment 3, *i.e.*, `map` parameters are passed by name and `let` is recursive. Assume that Jam supports the primitive `cons?` as the recognizer for non-empty lists. Consider the Jam expression:

```
let and := map x,y to if x then y else false;
     or := map x,y to if x then true else y;
 member := map x,l to
               and(cons?(l), or(x = first(l), member(x, cdr(l))));
   in member(1, cons(1, null))
```

(i) Using explicit substitution, show the *major* steps in the evaluation of this expression. Please use abbreviations to shorten your trace.

(ii) Assume that Jam passes parameters by value rather than by name. Show the major steps in the evaluation of the preceding expression. Please use abbreviations to shorten your trace.

Problem 6. (20 points)

Al Gaulle has designed the ultimate Algol dialect supporting passing parameters by value, by name, by reference, by result, and by value-result. For value-result parameter passing, assume that the argument evaluated once on entry to the procedure and that the resulting location is used on exit.

Consider the following Algol-like program (written in Java-like notation):

```
int i,j,a[5];     // a is an 5 element array with indices 0-4
void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}
for (j = 0; j < 5; j++) a[j] := j;

i := 1;
swap(i,a[i+1]);
write(i,a[2]);
```

What numbers does the program print if both parameters in `swap` are passed by:

1. value?
2. reference?
3. name?
4. value-result?

Algol evaluates procedure arguments in left-to-right order. You can get partial credit if you show your hand evaluation of the code. Some answers may be indeterminate.

Addendum Some additional potential topics for questions include:

- the basic properties of static and dynamic chains;
- simple questions the semantics of dynamic dispatch;
- questions about static distance coordinates
- (extra credit) questions about "domain theory" (continuity, monotonicity, etc.)