Comp 411: Sample Midterm Examination

(With Answers)

March 1, 2024

Name: ______________________________

Id #: ______________________________

NetID: ______________________________

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.

His extension ostensibly is a flawed design because the syntax as specified by the CFG above is ambiguous. There are two distinct parse trees for the following expression:

```
if flag1 then if flag2 then foo() else bar()
```

The else *clause can be associated with either of the two* if *expressions.*

```
if flag1 then (if flag2 then foo() else bar())
```

or

```
if flag1 then (if flag2 then foo()) else bar()
```

On the other hand, if we write the parser based on the corresponding syntax diagrams which maximize the length of each parsed sub-expression, the extra production for **`<if-exp>`** does not create any problems. The expression:

```
if flag1 then if flag2 then foo() else bar()
```

is parsed as

```
if flag1 then (if flag2 then foo() else bar())
```

because the **`else`** is absorbed by the inner **`<if-exp>`** in the longest match.

Problem 2. (10 points) Al Gaulle is responsible for maintaining a Jam (without recursive `let` as in Assignment 2) program written by another programmer. In the middle of the program, Al notices expression

```
let twice := map f to map t to f(f(t));
        t := 10;
in twice(map n to t*n)
```

Al decides to optimize the program by

(i) "inlining" the definition of **`twice`**,
(ii) reducing the application of the body [binding] of **`twice`** to **`map n to t*n`** [using beta-reduction], and
(iii) eliminating the now dead binding of **`twice`** to yield:

```
let t := 10;
in map t to (map n to t*n) ((map n to t*n) (t))
```

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

No. He did not optimize the program correctly.
The inlining of **`twice`** *is valid yielding*

```
=> let twice := map f to map t to f(f(t));
           t := 10;
   in (map f to map t to f(f(t))) (map n to t*n)
```

since **`twice`** *has no free variables. But the next step*

```
=> let twice := map f to map t to f(f(t));
           t := 10;
   in map t to (map n to t*n) ((map n to t*n) (t))
```

captures the free occurrence of **`t`** *in* **`map n to t*n`**.

[Optional:] *The correct implementation of the optimization uses safe substitution for* **`f`**

```
=> let twice := map f to map t to f(f(t));
           t := 10;
   in map s to (map n to t*n) ((map n to t*n) (s))
=> let t := 10;
   in map s to (map n to t*n) ((map n to t*n) (s))
```

which can be further optimized.

Problem 3. (20 points) Assume that our Jam dialect supports both raw **`let`** (as in Project 2) and **`letrec`** (the meaning of **`let`** in Project 3). Recall that the raw **`let`** expression

```
let x1 := E1; x2
   := E2; . . .
   xn := En;
in E
```

abbreviates

```
(map x1, x2, ..., xn to E)(E1, E2, ..., En)
```

and that **`letrec`** is the recursive generalization of **`let`** as described in Assignment 3. For this problem, we augment our Jam language with the **`let*`** construct from Scheme as defined below. The **`let*`** construct has the same syntax as `let` except for the change in the opening keyword (from **`let`** to **`let*`**). The **`let*`** construct can be defined in terms of the raw `let` construct as follows:

```
 let* x1 := E1; x2 := E2; ..., xn := En; in E
```

abbreviates

```
let x1 := E1;
in let x2 := E2;
   in ...
         in let xn := En;
            in E
```

For each of the following two Jam expressions (which could generate run-time errors):

1. circle each binding occurrence of a variable;
2. draw arrows from each bound occurrence back to the corresponding binding occurrence; and
3. draw a square box around any free occurrence of a variable within the entire expression.

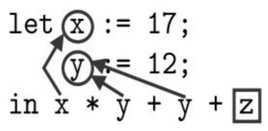
Do not classify Jam primitive operations including **`first`**, **`rest`**, **`cons`**, **`empty`**, **`empty?`**, **`cons?`**, **`list?`**, and all unary and binary operators as variables; they are function constants.

For example, given the expression

```
let x := 17;
    y := 12;
in x * y + y + z
```

the correct answer is:

For each of the two following expressions, circle each binding occurrence of a variable and draw arrows from each bound occurrence back to the corresponding binding occurrence.



```
let x := 17;
    y := 12;
in x * y + y + z
```

1.
```
let*
     fib := map n to
          letrec fibhelp := map k,fn,fnm1 to
                              if k = 0 then fn
                              else fibhelp(k - 1, fn + fnm1, fn);

                in if n < 2 then 1 else fibhelp(n – 1, 1, 1);

  fib100 := fib(100);

in fib100 * fib100 + fib(z)
```

2.
```
let pair := map x, y to
              let x := x;
                  y := y;
              // a functional representation of pairs
              in map msg to if msg = 0 then x else y;

in (pair(50, 100))(0)
```

Problem 4. (20 points) Let Jam have the name-value semantics specified in assignment 3, *i.e.*, **map** parameters are passed by name, **cons** is strict (as in Assignment 2 Jam), and **let** is recursive. 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, rest(l))));
  in member(1, cons(1, empty))
```

a. Using explicit substitution, show every step in the evaluation of this expression. Please use abbreviations to shorten your trace.

```
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, rest(l)));
in member(1, cons(1, empty))

=> let ... in (map x,l to ... )(1, cons(1, empty))
=> let ... in and(cons?(cons(1,empty)),
                  or(1 = first(cons(1,empty)), member(1,rest(cons(1,empty))))
=> let ... in (map x,y to if x then y else false)
                 (cons?(cons(1,empty)),
                        or(1 = first(cons(1,empty)), member(1,rest(cons(1,empty))))
=> let ... in if (cons?(cons(1,empty)) then
                or(1 = 1, member(1,rest(cons(1,empty))))
              else false
=> let ... in if true then
                 or(1 = first(cons(1, null)), member(1, rest(cons(1, null))))
              else false
=> let ... in or(1 = first(cons(1, null)), member(1, rest(cons(1, null))))
=> let ... in (map x,y to ...)(1 = first(cons(1, null)),
                              member(1, rest(cons(1, null))))
=> let ... in if 1 = first(cons(1, null)) then true
              else member(1, rest(cons(1, null)))
=> let ... in if 1 = 1 then true else member(1,rest(cons(1,empty)))
=> let ... in if true then true else member(1,rest(cons(1,empty)))
=> let ... in true
=> true
```

b. Assume that Jam passes parameters by *value* rather than by *name* and that **cons** is still strict (yielding value-value semantics from Project 3). Show every step in the evaluation of the preceding expression. Please use abbreviations to shorten your trace.

```
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, rest(l)));
in member(1, cons(1, empty))

=> let ... in (map x,l to ... )(1, cons(1, empty))
=> let ... in and(cons?(cons(1,empty)), or(1=first(cons(1,empty)), member(1,rest(cons(1,empty)))))
=> let ... in (map x,y to ... )
                (cons?(cons(1,empty)), or(1 = first(cons(1,empty)), member(1,rest(cons(1,empty)))))
=> let ... in (map x,y to ... )(true, or(1 = first(cons(1,empty)), member(1,rest(cons(1,empty)))))
=> let ... in (map x,y to ... )
                (true, (map x,y to if x then true else y)
                         (1 = first(cons(1,empty)), member(1, rest(cons(1,empty)))))
=> let ... in (map x,y to ... )(true, (map x,y to if x then true else y)
                                        (1 = 1, member(1,rest(cons(1,empty)))))
=> let ... in (map x,y to ... )(true, (map x,y to ... )(true, member(1, rest(cons(1,empty)))))
=> let ... in (map x,y to ... )(true, (map x,y to ... )(true, (map x,l to ... )
                                                          (1, rest(cons(1,empty)))))
=> let ... in (map x,y to ... )(true, (map x,y to ... )(true, (map x,l to ... )(1,empty)))
=> let ... in (map x,y to ... )(true,
                                (map x,y to ... )
                                   (true, and(cons?(empty), or(1 = first(empty), member(...)))))
=> let ... in (map x,y to ... )(true,
                                (map x,y to ... )
                                   (true, (map x,y to ...)
                                            (cons?(empty), or(1 = first(empty), member(...)))))
=> let ... in (map x,y to ... )(true,
                                (map x,y to ... )
                                   (true, (map x,y to ... )
                                            (false, or(1 = first(empty), member(...)))))
=> let ... in (map x,y to ... )(true, (map x,y to ... )
                                       (true, (map x,y to ... )
                                                (false,
                                                 (map x,y to if x then true else y)
                                                    (1 = first(empty), member(...)))))

=> run-time error: applying first to empty
```

Problem 5. (20 points) Al Gaulle has designed the ultimate Algol dialect supporting passing parameters by value, by name, by reference, and by value-result. For value-result parameter passing, follow the usual convention where the argument left-hand evaluated 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
procedure 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?

   Just prior to the **swap** *statement,* **a** = {0,1,2,3,4} *and* **i** = 1*. The call on* **swap** *has no effect on the actual parameters since they are passed by value. Hence, the program outputs:*

   1 2

2. reference?

   Just prior to the **swap** *statement,* **a** = {0,1,2,3,4} *and* **i** = 1*. The call on* **swap** *swaps the contents of* **i** *and* **a[2]***, yielding* a = {0,1,1,3,4} *and* i = 2*. Hence, the program outputs:*

   2 1

3. name?

   Just prior to the **swap** *statement,* **a** = {0,1,2,3,4} *and* **i** = 1*. Let us trace the call on* **swap** *in detail. Within the body of* swap*,* **x** *is synonymous with the variable* **i***;* **y** *is synonymous with the variable* **a[i+1]**. **temp** *is set to the contents of* **x** (**i**) *which is* 1. **x** (**i**) *is set to the contents of* **y** (**a[2]**) *which is* 2. **y** (**a[3]** *since* **i** *is now* 2*) is set* *to* **temp** *which is* 1*. On exit from* **swap***,* **a** = {0,1,2,1,4} *and* **`i`** = 2*. Hence, the program outputs:*

```
2 2
```

4. value-result?

   Just prior to the ***swap*** *statement,* **a** = {0,1,2,3,4} *and* **`i`** = 1*. After entering* ***swap,*** **x** *is bound to* 1 *and* **y** *is bound* 2 *just as in call-by-value. The left-hand value associated with* ***x*** *is the cell* **`i`** *and the left-hand value associated with* **y** *is the cell* **`a[2]`***. Just before exiting* ***swap****, the values of* **x** *and* **y** *have been swapped:* **x** = 2 *and* **y** = 1*. Call-by-result stores* 2 *in* **`i`** *and* 1 *in* **`a[2]`***. Hence, the program outputs:*

   ```
   2 1
   ```

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.

Problem 6. [20 points]

This problem uses value-value Jam dialect from Assignment 3 (recall that `let` is recursive). In this problem you will convert a simple Jam program from conventional syntax using named variables to `0`-based static-distance coordinate form. In this conversion, use

- the notation [**k**] to signify a sequence of *k* variables introduced by `map`, `let`, or `letrec`;
- the notation `(i:j)` for an occurrence of the static distance variable that is defined `i` levels outside the current (immediately enclosing) `map` or `let` construction and appears in the `j`th (`0`-based) position in the list of variables defined in the matching construction; and
- the right-hand-side expression E followed by a semi-colon for each binding `<var> :=` E; introduced in a `let` or `letrec` construction.

Hence, the static distance coordinate `(0:0)` in the body of a `map`, `let` or `letrec` construction refers to the `0`th (first) variable in the enclosing `map` or construction.

For example, the Jam program

```
let id := map x to x; in cons(id(17), empty)
```

has the `0`-based static distance representation:

```
let [*1*] map [*1*] to (0:0); in cons((0:0)(17), empty)
```

Note that the **:=** separator in `let` notation vanishes when it is converted to static-distance form, but we still need to put semicolons at the end of each binding expression to separate them. Also note that `0` as a static distance coordinate refers the *first* entity when counting. Hence, in the body of **`(map x to x)`**, **`x`** converts to that static distance coordinate **`(0:0)`**, *i.e.*, **`(map x to x)`** converts to **`(map [*1*] to (0:0))`**

Convert the following Jam program to `0`-based static distance form. Note that all variable names (*e.g.*, **`maplist`**, **`f`**, **`s`**, and **`x`**) will be replaced by static distance coordinates but the names of primitive operations (constants) like **`empty?`**, **`cons`**, **`first`**, and **`rest`** are retained.

```
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, rest(l))));
in member(1, cons(1, null))
```

Solution:

```
let [*3]
  map [*2] to if (0:0) then (0:1) else false;
  map [*2] to if (0:0) then true else (0:1);
  map [*2] to
    (1:0)(cons?((0:1)), (1:1)((0:0)=first((0:1), (1:2)((0:0),rest((0:1))
```

Addendum On the midterm exam, there may be an extra-credit question involving domain theory.