Comp 411: Sample Midterm Examination

March 1, 2022

Name: _____

Id #: _____

NetID: _____

Instructions

1. The examination is closed book. Fill in the information above and the pledge below.
2. There are 6 problems on the exam, totaling 100 points on the exam.
3. 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 Jam 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?

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 Project 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 ifollowing two Jam expressions (which could generate run-time errors):

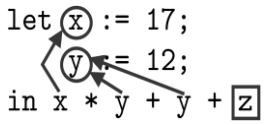
a. circle each binding occurrence of a variable;
b. draw arrows from each bound occurrence back to the corresponding binding occurrence; and
c. draw a square box around any free occurrence of a variable within the entire expression.

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:



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;

              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 Project 3, *i.e.*, **map** parameters are passed by name, **cons** is strict (as in Project 2 Jam and Scheme/Racket), 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, null))
```

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

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.

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?

2. reference?

3. name?

4. value-result?

Note: 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 Projectr 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 jth (`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 to signify each binding `<var> :=` E; introduced in a (recursive) `let` 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 the `let` notation for a local binding vanishes when it is converted to static-distance form, but we still need to specify the right-hand-side expression followed by a semicolon as a separator. 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.*, **`and or member`**, **`x`**, **`y`**, and **`l`**) are 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))
```

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