Spring 1997 CIS 624

Final Exam

June 11, 1997

Name: ______________________________

Java Statements and Expressions (15 points)

```
class A {}

class Test {
  public static void main (String args[]) {
    A y = new A();
    boolean b = true;
    while (true) {
      A x = new A();
      if (b) {
        y=x;
        b=false;
        /* XXX */
        continue;
      }
      if (x == y) System.out.println("equal");
      else System.out.println("not equal");
      break;
    }
    /* YYY */
  }
}
```

Given that the *value* of the expression `new A()` is a newly allocated *location* that contains `<objectA>`:

- What does the program print?

- Justify your answer by showing the contents of both the environment and the store every time the execution reaches the points marked `XXX` and `YYY`.

Closures (20 points)

I have mentioned once in class that *everything* can be encoded in the lambda calculus. Here is an encoding of numbers using higher-order functions (due to Church):

```
(define simpleCN (lambda (n) (lambda (f x) (iterate f n x))))
```

The specification (which is enough to infer the definition of `iterate`) is that `simpleCN` takes a number `n` and encodes it as follows:

- 0 is encoded as the function `(lambda (f x) x)`,
- 1 is encoded as the function `(lambda (f x) (f x))`,
- 2 is encoded as the function `(lambda (f x) (f (f x)))`,
- 3 is encoded as the function `(lambda (f x) (f (f (f x))))`

and so on. The usual arithmetic operations can be defined on encoded numbers:

```
(define succ (lambda (m) (lambda (f x) (f (m f x)))))
(define add (lambda (m n) (lambda (f x) (m f (n f x)))))
(define mult (lambda (m n) (lambda (f x) (m (lambda (x) (n f x)) x))))
(define toString (lambda (m) (number->string (m add1 0))))
```

Your job is to complete the following Java implementation of Church numerals:

```
class Test {
  public static void main (String args[]) {
    ChurchNumeral m = new SimpleCN(5);
    ChurchNumeral n = new SimpleCN(8);
    System.out.println(m);          // uses the toString method to print 5
    System.out.println(m.succ());   // prints 6
    System.out.println(n);          // prints 8
    System.out.println(n.add(m));   // prints 13
    System.out.println(n.mult(m));  // prints 40
  }
}

abstract class ChurchNumeral {
  abstract Object apply (Closure f, Object x);
  ChurchNumeral succ () { return new SuccCN(this); }
  ChurchNumeral add (ChurchNumeral m) { return new AddCN(this,m); }
  ChurchNumeral mult (ChurchNumeral m) { return new MultCN(this,m); }
  public String toString () {
    Integer res = (Integer)apply(new add1Closure(), new Integer(0));
    return res.toString();
  }
}
```

```
class SimpleCN extends ChurchNumeral {
  int n;
  SimpleCN (int n) { this.n = n; }
  Object apply (Closure f, Object x) { return f.iteratedApply(n,x); }
}
```

Exceptions (15 points)

Even though Scheme does not have handleable exceptions, they are definable in the language. Write Scheme definitions for `raise` and `handle` so that the following code works as shown:

```
(define listLookup
  (lambda (nlist index)
    (printf "Looking up position ~s in ~s~n" index nlist)
    (if (null? nlist)
        (raise)
        (if (= index 0)
            (car nlist)
            (listLookup (cdr nlist) (sub1 index))))))

(define newLookup
  (lambda (nlist index)
    (handle
     (lambda () (listLookup nlist index))
     (lambda () (newLookup nlist (quotient index 2))))))

> (newLookup '(99 88 77 66) 2)
Looking up position 2 in (99 88 77 66)
Looking up position 1 in (88 77 66)
Looking up position 0 in (77 66)
77

> (listLookup '(99 88 77 66) 5)
Looking up position 5 in (99 88 77 66)
Looking up position 4 in (88 77 66)
Looking up position 3 in (77 66)
Looking up position 2 in (66)
Looking up position 1 in ()
Error in raise: Unhandled exception.

> (newLookup '(99 88 77 66) 5)
Looking up position 5 in (99 88 77 66)
Looking up position 4 in (88 77 66)
Looking up position 3 in (77 66)
Looking up position 2 in (66)
Looking up position 1 in ()
Looking up position 2 in (99 88 77 66)
Looking up position 1 in (88 77 66)
Looking up position 0 in (77 66)
77
```

```
> (newLookup '(99 88 77 66) 9)
Looking up position 9 in (99 88 77 66)
Looking up position 8 in (88 77 66)
Looking up position 7 in (77 66)
Looking up position 6 in (66)
Looking up position 5 in ()
Looking up position 4 in (99 88 77 66)
Looking up position 3 in (88 77 66)
Looking up position 2 in (77 66)
Looking up position 1 in (66)
Looking up position 0 in ()
Looking up position 2 in (99 88 77 66)
Looking up position 1 in (88 77 66)
Looking up position 0 in (77 66)
77
```

Programming Style (15 points)

A set of integers is much the same as a set of characters, sorting strings is much the same as sorting floats. *Polymorphism* provides a general approach to describing data or algorithms where the structure is independent of the type of the element manipulated. As a simple example, consider an algorithm for swapping a pair of elements, both of the same type. One would like to write code such as the following:

```
class Pair<elemType> { // not valid Java code
  elemType x, y;
  Pair (elemType x, elemType y) { this.x=x; this.y=y; }
  void swap () {
    elemType t;
    t=x;
    x=y;
    y=t;
  }
}

class Test {
  public static void main (String args[]) {
    Pair<String> p = new Pair("world!", "Hello, ");
    p.swap();
    System.out.println(p.x + p.y); // should print Hello, world!

    Pair<int> q = new Pair(22, 64);
    q.swap();
    System.out.println(q.x - q.y); // should print 42
}
```

Unfortunately such code is not valid in Java. Instead Java programmers have two choices: either duplicate the declaration of the class `Pair` once for the `String` use and once for the `int` use, or cast everything to and from `Object`. Write both solutions and discuss in no more than 3 lines the disadvantages of each approach to simulating polymorphism.

Inheritance (20 points)

For each `println` statement below, write the value printed:

Program I

```
class A {
  static String s1 () { return "Good "; }
  String s2 () { return "dog"; }
}

class B extends A {
  static String s1 () { return "Bad "; }
  String s2 () { return "cat"; }
}

class Test {
  public static void main (String[] args) {
    A obj = new B();
    System.out.println(obj.s1() + obj.s2()); // .........................
  }
}
```

Program II

```
class S {
  int x = 0;
  int getx() { return x; }
}

class T extends S {
  int x = 1;
  int getx() { return x; }
  int sx() { return super.x; }
  int getsx() { return super.getx(); }
}

class Test {
  public static void main (String[] args) {
    T objOne = new T();
    System.out.println(objOne.x);       // ............................
    System.out.println(objOne.getx());  // ............................
    System.out.println(objOne.sx());    // ............................
    System.out.println(objOne.getsx()); // ............................

    S objTwo = new S();
```

```
    System.out.println(objTwo.x);       // ...........................
    System.out.println(objTwo.getx());  // ...........................

    objTwo = objOne;
    System.out.println(objTwo.x);       // ...........................
    System.out.println(objTwo.getx());  // ...........................
  }
}
```

Program III

```
class A {
  int i;
  A () { i = 1; }
  int geti() { return(i); }
  int m () { return (this.geti()); }
}

class B extends A {
  int i;
  B () { i = 2; }
  int geti() { return(i); }
  int m () { return (super.m()); }
}

class C extends B {
  int i;
  C () { i = 3; }
  int geti() { return(i); }
  int m () { return (super.m()); }
}

class Test5 {
  public static void main (String args[]) {
  C obj = new C();
  System.out.println(obj.m());          // ...........................
  }
}
```

Coroutines (15 points)

Write the function `two-in-a-row?` that behaves as follows:

```
> (define tree1 '(((1 2) (3 4)) (5 6)))
> (define tree2 '(((1 2) (3 4)) (4 5)))
> (two-in-a-row? tree1)
Comparing 1 and 2
Comparing 2 and 3
Comparing 3 and 4
Comparing 4 and 5
Comparing 5 and 6
Comparing 6 and 0
#f
> (two-in-a-row? tree2)
Comparing 1 and 2
Comparing 2 and 3
Comparing 3 and 4
Comparing 4 and 4
#t
```

In other words, the function `two-in-a-row?` takes a binary tree and checks if any two consecutive leaves in the fringe are equal. Your solution *must* reuse parts of the solution for the `sameFringe` code given below and must be such that the traversal of the tree stops as soon as the answer is determined.

```
(define make-generator
  (lambda (tree)
    (letrec ((caller '*)
             (generate-leaves
              (lambda ()
                (letrec ((loop
                          (lambda (tree)
                            (if (number? tree)
                                (call/cc (lambda (genrest)
                                   (set! generate-leaves
                                         (lambda () (genrest '*)))
                                   (caller tree)))
                                (begin (loop (car tree))
                                       (loop (cadr tree)))))))
                  (loop tree)))))
      (lambda ()
        (call/cc (lambda (k)
                   (set! caller k)
                   (generate-leaves)
                   (caller 0)))))))
```

```
(define driver
  (lambda (gen1 gen2)
    (let ((leaf1 (gen1)) (leaf2 (gen2)))
      (if (= leaf1 leaf2)
          (if (zero? leaf1)
              #t
              (driver gen1 gen2))
          #f))))

(define sameFringe
  (lambda (tree1 tree2)
    (let ((gen1 (make-generator tree1))
          (gen2 (make-generator tree2)))
      (driver gen1 gen2))))
```