Final Exam

June 13, 1996

General Information

The exam consists of 5 problems worth 100 points and an extra credit worth 5 points.

- Name:
- Id:

Good luck!

Problem 1: Exceptions (20 points)

- (10 points) Explain why exception handlers should be dynamically scoped?

- (10 points) What are the values of the following expressions:

```
newexception E in
  tryhandle E (proc (x) 100)
  in 5
```

```
newexception E in
  tryhandle E (proc (x) 100)
  in (proc (y) (raise E y))
```

```
newexception E in
  tryhandle E (proc (x) +(x,1))
  in tryhandle E (proc (x) (raise E +(x,2)))
     in (raise E 3)
```

```
newexception E in
 tryhandle E (proc (x) +(x,1))
 in tryhandle E (proc (x) (raise E +(x,2)))
    in let f = proc (x) (raise E x)
       in let g = tryhandle E (proc (x) (raise E +(x,100)))
                  in proc (y) f(y)
          in g(4)
```

Problem 2: Continuations (20 points)

- (5 points) Evaluate the following Scheme expressions:

```
(add1 (call/cc (lambda (k) (sub1 (sub1 (k 3))))))
==>


(define cont '*)
(add1 (call/cc (lambda (k) (begin (set! cont k) 3))))
==>

(cont 9)
==>

(cont 7)
==>
```

- (15 points) Define the Scheme procedure `call-once/cc` that behaves like `call/cc` except in the following way. If we capture a continuation using `call-once/cc` then this continuation can only be invoked once! Any other attempt to invoke the continuation should signal an error. For example, replacing `call/cc` by `call-once/cc` in the above expressions should not change any results except that `(cont 7)` should signal an error.

Problem 3: CPS (30 points)

Consider the following recursive function:

```
(define count-occ
  (lambda (key lst)
    (if (null? lst)
        0
        (let ([elem (car lst)])
          (if (eqv? key elem)
              (add1 (count-occ key (cdr lst)))
              (count-occ key (cdr lst)))))))
```

- (10 points) Convert the definition to CPS using the usual representation of continuations as procedures.

- (10 points) Represent continuations using records.

- (10 points) Show the invariants that must be satisfied by any representation of continuations. Represent continuations using numbers and prove that the new representation satisfies the invariants.

Problem 4: Inheritance (15 points)

In Java the keyword `super` is available in all non-static methods of an extended class. For example, the following code prints `B B A`:

```
class A {
  String getid () { return "A "; }
}

class B extends A {
  String getid () { return "B "; }
  String test () {
    A sr = super;
    return ( this.getid() + sr.getid() + super.getid() );
  }
}

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

Note that `super` is used in two different ways. Our interpreter can already handle uses of `super` in method invocations like `super.getid()`. We are now interested in extending the interpreter to handle the other use of `super`.

We choose the following syntax:

```
letsuper var in exp
```

where we make the scope of the binding explicit. The informal semantics of the new expression is to lexically bind `var` in `exp` such that any occurrence of `var` in `exp` acts as a reference to the current object as an instance of its superclass. Here is an attempt to formalize this semantics using our interpreter:

```
(define eval
  (lambda (exp env class inst)
    (variant-case exp
      [letsuper (var body)
         (let* ([newinst (make-instance
                          (class->parent (instance->class inst))
                          (instance->i-vals inst))]
                [newenv (extend env (list var) (list (box newinst)))])
           (eval body newenv class inst))]
      ...)))
```

Explain why the above code is *wrong*! Your explanations should be supported by concrete examples that show how the interpreter evaluates things differently from the intented Java semantics.

Problem 5: More Inheritance (15 points)

Consider the following Java classes and declarations:

```
class A {
  static int avar = 1;
  int getvar () { return avar; }
  int getvartoo () { return getvar(); }
}

class B extends A {
  static int bvar = 2;
  int getvar () { return bvar; }
  int getvartoo () { return super.getvartoo(); }
}

A object_1 = new A ();
A object_2 = new B ();
```

- (10 points) We have an object `XXX` of type `A` that could either be `object_1` or `object_2`. Which of the following operations would you use to determine the identity of `XXX`. Justify your answer for *each* operation:
  - `XXX.avar`
  - `XXX.bvar`
  - `XXX.getvar()`
  - `XXX.getvartoo()`
- (5 points) Can we optimize the above code by eliminating the methods `getvartoo` from both classes and using `getvar` instead?

Extra Credit (5 points)

Assume two libraries (a C++ version and a Java version) that contain a definition for a File object. The File object has two `read` methods: a `public` one (callable by anyone) and a low level `private` one (only callable by the object's methods). The intention is that the public `read` first performs security checks and then calls the private `read`.

Determine whether each of the following C++ features is a security hazard. In other words, does the feature enable a malicious C++ user to get access to the private `read` method without going through the security checks of the public method? (The malicious code does not have to be portable!)

- Unchecked cast operations in C++ vs checked cast operations in Java.

- Pointer arithmetic in C++ vs implicit pointers in Java.

- Destructors in C++ vs garbage collection in Java.