1. **(15 points)** .................................................... Memento Mori

Here is some JavaScript code which takes a function as an argument and transforms it into a memoized version. In this particular case, the memoized function remembers the last invocation, so that if you call the function twice with the same argument, it returns the cached result.

```javascript
var memo = function __impl_memo(f) {
    var saved_x;
    var saved_v;
    return function __impl_inner(x) {
        if (saved_x != x) {
            saved_x = x;
            saved_v = f(x);
        } else {
            // ***
        }
        return saved_v;
    };
};

var g = memo(function __impl_addone(x) { return x + 1; });
g(1);
g(1);
```

**(a) (15 points)** While executing this function, six bindings are created: memo, g, f, saved_x, saved_v and x. Assuming that execution is at the marked *** line, write the current values of each of these bindings in the correct activation records, assuming (3) is the current activation record. If a variable is bound to a closure, specify which closure using M, I or A, and fill in the environment pointer of the closure by writing (1), (2) or (3).

<table>
<thead>
<tr>
<th>Activation Records</th>
<th>Closures</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) top access link (none)</td>
<td>M: (____, code for memo)</td>
</tr>
<tr>
<td>(2)</td>
<td>I: (____, code for inner)</td>
</tr>
<tr>
<td>(3) access link (2)</td>
<td>A: (____, code for addone)</td>
</tr>
</tbody>
</table>
2. (15 points) Tail Recursion, Exceptions and Continuations

In this question we’re going to explore the use of continuations.

(a) (6 points) Recall that we can use continuation to translate a recursive function into one that is tail recursive. This is called continuation passing style (CPS).

Convert the following recursive function \texttt{recur} into a tail recursive function.

```javascript
function recur(n) {
    if (n == 0)
        return 1;
    else
        return n*recur(n-1)+1;
}
```

Fill in the rest of the code below. Here \( k \) is the continuation passed into \texttt{recur}.

```javascript
function recur(n, k) {
    if (n == 0)
        \_________________________; 
    else
        \_________________________; 
}
```

(b) (3 points) Recall that tail-recursive functions can be transformed into while-loops. If we allocate closures on the stack, using \( \text{big-O} \) notation, express the overall size of the stack used when computing \texttt{recur(n)} with tail-recursion optimization enabled. Briefly justify your answer.

(c) (6 points) The \texttt{recur} function defined above will not terminate if the argument is negative. Let’s use JavaScript exceptions to handle this exceptional behavior:

```javascript
function recur(n) {
    if (n < 0)
        throw new Error("Invalid argument!");
    if (n == 0)
        return 1;
    else
        return n*recur(n-1)+1;
}
```
This function throws an exception when the argument is invalid. Using this, we can now define a safe version of `recur` that simply returns a default value when the argument is invalid:

```javascript
function safeRecur(n, d) {
    try {
        return recur(n);
    } catch (e) {
        return d;
    }
}
```

Recall that continuations can be used to encode exceptions. Hence, the `try/catch` blocks can be seen as a syntactic sugar; we can desugar these blocks when transforming a function to CPS by passing in an additional continuation. Rewrite `recur` and `safeRecur` in CPS without using `try`, `catch`, `throw`.

(Hint: The anonymous function that we have partially defined for you in `safeRecur` is used in place of the `throw` function.)

```javascript
function recur(n, k, throwFunc) {
    if (n < 0) {
        ________________;
    } else if (n == 0) {
        /* code from (b), line 3 */
    } else {
        recur(n-1, /* code from (b), line 4 */, ________________);
    }
}

function safeRecur(n, d, k) {
    recur(n, k, function(x){______________});
}
```
3. (20 points) ..................... Multiple Inheritance and Variance

Considering the following class definition, and code using it:

```cpp
class A {
    public:
        virtual void g();
        int x1;
        int x2;
};
class B {
    public:
        int y;
        virtual B* f();
        virtual void h();
};
class C : public A, public B {
    public:
        int z;
        virtual C* f();
};

C *pc = new C;  B *pb = pc;  A *pa = pc;
```

(a) (10 points) Fill in the object layout and vtables for a C object. For each entry in the vtable, indicate the function to which the slot points and any deltas. Use δ_B to represent the distance between pb and pa.

```
Object       Virtual function tables
pc, pa →  vpctr →
          |        |
            |        |
            |        |
pb →  vpctr →
          |        |
            |        |
            |        |
```

(b) (5 points) C++’s “covariant return type” rule states that a method can be overridden with a new method that returns a subtype of the original return type. This rule makes it acceptable for C::f to return a C* while B::f returns a B*. Consider the following virtual method calls:

```cpp
B *pb;
B *new_pb = pb->f();
```

What is the necessary return offset that must be applied to the value returned by the implementation of f when:

i. pb is an instance of B?
ii. pb is an instance of C?

iii. True or False: the return offset of a call to f is statically known. (Circle one.)

(c) (5 points) The C++ fairy came and removed all of the virtual keywords from your class definition. Consider the same piece of code:

B *pb;
B *new_pb = pb->f();

What implementation of f is called when:

i. pb is an instance of B?

ii. pb is an instance of C?

iii. True or False: the return offset of a call to f is statically known. (Circle one.)
4. (30 points) Activation Records and Garbage Collection

The activation record model we studied in class assumed infinite memory; however, it is not too difficult imagine garbage collecting activation records. Specifically:

- The root set consists of the activation records on the stack (as defined by control links, though in this problem we won’t treat these explicitly).
- Access links and the pointers in fields of the activation record define what the set of live data is.

(a) (12 points) Fill in the activation records and closures corresponding to the state of this JavaScript program when it reaches the line marked with ***. There may be more blanks than are strictly necessary. (Don’t worry about the refcount field for now; it will be utilized in the next part of the question.)

```javascript
var z = 1
function f(x) {
  function g(y) {
    // ***
    return x + y + z;
  }
  return g;
}

f(2)(3);
```

(b) (10 points) Ben Bitdiddle suggests that activation records could be garbage collected using reference counting. Specifically, each activation record AND closure is associated with a reference count, which tracks the incoming access links, pointers, and membership in the root set (which counts as “one” incoming pointer).

In the previous JavaScript program, when execution reaches the line marked ***, the root set consists of ALL three activation records (e.g., ignoring all pointers, the reference count of each activation would be one). This statement is wrong; only two of the activation records are live. We haven’t fixed the problem solution and the model solution yet.

i. (5 points) Annotate each activation record and closure with its reference count at this point in time. (Hint: no object has a refcount of 0.)

ii. (2 points) One of your activation frames has a pointer to a closure for g. Once execution returns from the function calls (e.g. after the call to f), what is the reference count on this activation frame?
iii. (3 points) In a sentence, explain why Ben Bitdiddle reference counting scheme, as stated, is a bad idea.

(c) (8 points) Here is an implementation of delay and force from the first lab.

```javascript
function delay(f) {
    var is_empty = true;
    var saved;
    return function () {
        if (is_empty) {
            saved = f();
            is_empty = false;
        }
        return saved;
    };
}

function force(t) {
    return t();
}
```

Consider the snippet `var x = delay(someFunction); force(x);`. Suppose that after executing this code, `x` is part of the root set (and will get forced again later in the program), but there are no other references to `someFunction` elsewhere in the program. Circle the correct answer for the following questions:

i. True or False: `someFunction` causally influences the rest of the program.

ii. True or False: Under reference counting (e.g., using Ben’s scheme), `someFunction` will be garbage collected.

iii. True or False: Under tracing collection, `someFunction` is considered live.

iv. True or False: This function is correctly implemented, with regards to both correctness and performance.
5. (20 points) Declaration-site variance inference

Josh Bloch famously stated that “We simply cannot afford another wildcards”, referring to the complexity of use-site variance stemming from wildcards. Accordingly, you’ve been assigned to a Java task force to get rid of use-site variance and add declaration-site variance to Java. However, in a shocking break from historical precedent, the Java committee believes that Java programmers should not have to explicitly write the variance of their definitions. Instead, they want to infer variance automatically!

(a) (1 point) What should the inferred variance of this interface be? 

```java
interface F<A> {
    void put(A x);
}
```

(b) (3 points) What should the inferred variance of this interface be? 

```java
interface G<A> {
    void write(A x);
    F<A> getF();
}
```

(c) (3 points) What should the inferred variance of this interface be? 

```java
interface H<A> {
    void oneF(F<A> x);
    void twoF(F<F<A>> x);
}
```

(d) (5 points) The committee asks you to implement a prototype declaration-site variance for Java. However, you’re too lazy to actually modify a Java compiler, and decide to write a source-to-source translator, to translate declaration-site variance to use-site variance. If you infer that an interface M<T> is covariant, how should you rewrite code that uses this interface?

(e) (8 points) The developers of the JScience library are a bit worried about your variance inference proposal. In their library, they have implemented a number of classes which are parametrized by a “unit of measure”; for example, Unit<Mass> indicates a scalar quantity which measures mass. In the definition of Unit<Q>, Q is never actually used in the body of the class; the type parameter is used purely to help clients avoid mixing up their units. What variance(s) will you infer, and why is that a bad thing?

(f) Extra credit. (3 points) What should the inferred variance(s) of this interface be? 

```java
interface Q<A> {
    Q<Q<A>> f();
}
```
6. (25 points) ............................. Implementing MVars with STM

MVars can be implemented using STM using this definition:

```haskell
type MVar a = TVar (Maybe a)
```

Using the following STM functions (and the fact that STM is a monad):

```haskell
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
retry :: STM a
orElse :: STM a -> STM a -> STM a
```

implement the following MVar operations. If you have forgotten Haskell syntax, we’ll accept pseudocode, but you must explicitly say so and be CLEAR about how your implementation uses the STM primitives.

(a) (5 points)

```haskell
-- | Allocates a new MVar filled with a value
newMVar :: a -> STM (MVar a)
newMVar x =
```

(b) (10 points)

```haskell
-- | Puts a value into an MVar, blocking if it is already full.
putMVar :: MVar a -> a -> STM ()
putMVar m x =
```

(c) (10 points) A user on the GHC bug tracker suggested that there should be some mechanism to wait on multiple MVars, so that a value can be put into whichever one that is empty first, preferring the first one if both are empty. You may assume a correctly implemented putMVar implementation.

```haskell
-- | Put a value into the first empty MVar, blocking if both
-- are full. Prefers the first MVar if both are empty.
putTwoMVar :: MVar a -> MVar a -> a -> STM ()
putTwoMVar m1 m2 x =
```