Question 1. (4 points) What are the values of the following Scheme expressions

(a)
```
(define x 1)
(define y 2)
(define z 3)
(let ((x 3)
      (y (+ x 2))
      (z (+ x y 5)))
    (* x z))
```

24

(b)
```
(define x 1)
(define y 2)
(define z 3)
(let* ((x 3)
      (y (+ x 2))
      (z (+ x y 5)))
    (* x z))
```

39

Question 2. (4 points) The let construct is not a fundamental Scheme primitive because it can be defined without having it built in to the language. Give a lambda expression that has the same effect as the following let expression:

```
(let ((v1 e1)
      (v2 e2))
   x)
```

((lambda (v1 v2) x) e1 e2)

Question 3. (3 points) We looked at several parameter passing methods that had different rules for evaluation. Two in particular were call-by-name (thunks) and call-by-need. What is the key difference between these two?

Call-by-name reevaluates the thunk each time the parameter value is referenced in the called function. Call-by-need, in contrasts, evaluates the parameter the first time it is referenced, then saves the answer to reuse if the parameter is referenced again.

[Notes: If evaluating the parameter is side-effect free, the two parameter-passing methods are semantically equivalent. Reevaluating the thunk each time the parameter is referenced can be expensive, but call-by-need has some bookkeeping overhead, which can matter if evaluating the parameter is cheap.]

Question 4. (6 points) Recall that in scheme, the function call/cc evaluates a function passing it the current continuation as an argument (i.e., (call/cc (lambda (k) e)). Use call/cc to write a program that loops indefinitely printing the sequence of integers starting from 0 (i.e., 0, 1, 2, 3, …). *Do not* use recursive procedures or assignments.

Here are a couple of solutions. Both work but the second might be a bit easier to understand (and also prints the output a bit more legibly by not running all the numbers together).

```
((lambda (foo)
   (begin (print (car foo))
          ((cdr foo) (cons (+ 1 (car foo)) (cdr foo)))))
       (call/cc (lambda (k) (cons 0 k))))


(apply (lambda (k x)
         (begin (print x) (newline)
                (k (list k (+ x 1)))))
       (call/cc (lambda (cc) (list cc 0))))
```

Question 5. (6 points) Two possible properties of type systems, like all formal logic systems, are whether they are *sound* and/or *complete.*

(a) What does it mean for a type system to be sound?

A type system is sound if it never makes a false claim that something will work correctly. It's ok for it to not be able to decide whether something is correct, but it is not sound if something that is claimed to be correct fails during execution.

(b) What does it mean for a type system to be complete?

If a type system is complete, it will always be able to give a positive or negative answer about whether something is properly typed.

[It's not just sufficient to say that all correct programs are accepted – you also need the requirement that all incorrect programs are rejected. Otherwise, a type system that accepted everything unconditionally (always answers "ok") would be considered to be complete.]

Question 6. (3 points) One of Java's typing rules is that if A is a subtype of B, then the type of an array holding A objects (A[]) is a subtype of the array type that holds B objects (B[]). In other words, A<:B implies A[]<:B[].

If a Java type checker verifies that all array types in a program are used properly with this rule, are we guaranteed that there will be no type errors involving arrays during execution as things are inserted and removed? Give a brief argument if this is true, or a short example that demonstrates why this is not necessarily the case.

No, this is not sound. The following code will typecheck, but will lead to an exception being thrown during execution.

```
String[] strings = new String[1];
Object[] objects = strings;         // OK, String[] <: Object[]
objects[0] = new Integer(1);        // typechecks ok, an
                                    //  Integer is an Object,
                                    //  but fails at runtime
```

[Test taking hint: several people just gave an example, leaving it to the grader to figure out if that implied yes or no. It's good practice to answer the question completely.]

Question 7. (6 points) Write a Smalltalk method **do:while:** that works like the do-while loop in C/C++/Java. That is

do: aBlock **while:** aCondition

should repeatedly execute aBlock as long as aCondition is true. The condition should be evaluated *after* each execution of aBlock, which, in particular, means that aBlock will always be executed at least once, even if the condition is false the first time it is evaluated.

In addition, and unlike the C/C++/Java do-while loop, each time the loop body, aBlock, is executed, it should be passed an integer parameter that is initialized to the number of times the loop has iterated, i.e., the parameter should be 1 when aBlock is executed the first time, 2 the next time, and so forth. The count should be reset to 1 each time the **do:while:** method begins execution, i.e., the count should should not carry over from one **do:while:** loop to another.

```
do: aBlock while: aCondition
   | x |
   x := 1.
   aBlock value: x.
   aCondition whileTrue: [
      x := x+1.
      aBlock value: x.
   ].
```

Question 8. (6 points) (a) Depth subtyping relates the types of different records (or objects). When it [x1:t1, …, xi:ti, …, xn:tn] a subtype of [x1:t1, …, xi:tj, …, xn:tn] under the depth subtyping rule? (i.e., assuming that all of the types are the same except for ti and tj).

When ti is a subtype of tj (ti<:tj).

(b) When is a function type t1->t2 a subtype of function type t3->t4?

t1->t2 <: t3->t4 if t2 <: t4 and t3 <: t1.

Question 9. (6 points) Different type systems have different notions of when two types are equivalent. A key distinction is between type systems that distinguish between name (nominal) equivalence and structural equivalence.

(a) Give a brief explanation of the difference between these notions of type equivalence.

Structural subtyping defines two types to be equivalent if they contain the same number of fields and the fields have equivalent types. Name equivalence says that two differently named types are different, even if they have the same fields with the same types.

(b) Give one example or reason why name (nominal) equivalence would be better than structural equivalence.

A good example is polar and rectangular complex numbers. These are likely to have the same fields (a pair of floating-point numbers) and operations. Under those circumstances a structural subtype system would say they are the same type, while a named type system would prevent logic errors caused by combining numbers of the different types without proper conversions.

(c) Give one example or reason why structural equivalence would be better than name (nominal) equivalence.

Structural equivalence allows two types that are logically equivalent (perhaps developed at different times in different locations) to be used together without need for modifying or rewriting the code.

Question 10. (3 points) Both overloading and multimethods give us the ability to define different methods with the same name but different numbers or types of parameters. What is the key difference between them?

Normal overloading uses the dynamic type of the message receiver (`self` or `this`) to decide which class defines the set of methods that can be called. The other argument types are used to resolve at compile time which method from the selected class to use.

With multimethods, the dynamic types of all of the arguments as well as the type of the receiver are used to decide during execution which method to call.

Question 11. (6 points) (a) A garbage collector needs to be able to locate all objects that are *reachable* or in use. Give a brief, but precise description of how a garbage collector can find all objects that are currently reachable.

The key initial operation is to identify all of the objects that can be reached directly from the *root set* – all global or static variables or environments, plus the set of variables in all active methods or functions. Then those objects are checked and all objects reachable from them are identified. This continues until no new reachable objects can be found.

[For full credit, your answer had to specifically describe where the search starts, not just "find some objects and follow the references to other objects".]

(b) Fans of garbage collection sometimes claim that "memory leaks" (i.e., memory that is allocated but no longer being used) are impossible if a good garbage collector is used. Is this true? If so, give a brief argument as to why, if not, give an example showing why not.

Even with a good garbage collector, it is possible for no-longer in use data to be reachable if it can be accessed directly or indirectly from the root set. A fairly common cause of this is to have a static or class variable that retains a reference to an unneeded object. Because it is reachable from the root set it cannot be reclaimed by the garbage collector.

[One student's answer had a great take on this: "the garbage collector cannot read the programmer's mind"!]

Question 12. (7 points) Consider the following Scheme program:

```
(define (map lst f)
  (if (null? lst)
      ()
      (cons (f (car lst)) (map (cdr lst) f))))

(let ([x 0])
  (map '(1 2 3 4) (lambda (y)
                    (begin (set! x (+ x 1))
                           (+ x y)))))
```

Convince yourself that this evaluates to the list (2 4 6 8). This effect is possible because the function passed to map refers to and mutates the free variable x. As it turns out, it's always possible to convert a program that makes reference to free variables into one that doesn't. This process is called closure conversion. In short, closure conversion has four steps:

- Add an extra argument to all lambdas to represent the environment. Use an assocation list to represent the environment, much as in homework 5.
- Change references to free variables to instead look up (or mutate) the variable in the environment.
- Change all lambda expressions to actually return a pair of the lambda and the current environment.
- Change all function invocation expressions to fetch the function and environment from the pair and invoke the function with the environment as the final argument.

In essence, this process converts the program into one that keeps track of its own environments. Some interpreters do this automatically before executing a program and then only implement a subset of the language that has no free variables.

Convert the above code using the closure conversion process so that functions only make reference to their own arguments or top-level bindings, never free variables. Use the skeleton code provided below. Hint: the environment will be represented by an association list, so use the built-in assoc function to simplify your task.

```
(define (map lst f)
  (if (null? lst)
      ()
      (cons (_(car f)_ (car lst) _(cdr f)_) (map (cdr lst) f))))

(map '(1 2 3 4) (cons
                 (lambda (__y env__)
                    (begin

          (__set-cdr! (assoc 'x env) (+ (cdr (assoc 'x env)) )__)

                           (_+ (cdr (assoc 'x env)) y_)))

                 (list __(cons 'x 0)__)))
```