Name _____ Section _____

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-notes
- Please stop promptly at 10:20
- There are a total of 60 points, distributed unevenly among the questions
- Please try to write neatly – style matters, but we'll take into account the fact that this is a short exam and it's not always possible to have the time to revise and clean up everything.
- Please keep your answers brief and to the point.

Advice:

- Read questions carefully and understand what's asked before you start writing.
- Leave evidence of thoughts and intermediate steps so you can get partial credit.
- Skip around – if you get hung up on a question, try the next one and come back.
- If you have questions, ask – raise your hand and someone will come to your seat and try to help you out.
- Relax. You are here to learn.

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))
```

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

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)
```

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?

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.

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?

(b) What does it mean for a type system 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.

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.

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).

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

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.

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

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

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?

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.

(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.

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 lst) __________) (map (cdr lst) f))))

(map '(1 2 3 4) (cons
                 (lambda (_________)
                    (begin

(_____________________________________________________________________)

                             (_____________________________________)))

                 (list _______________)))
```