Networks

1a. Define a `latency` procedure that takes one input, a list of lists of bit arrival times, and produces as output a number representing the latency of the communication. For example, imagine that we send three bits across four semaphore stations. The first bit reaches the first station at 1 second, the second station at 3 seconds, the third station at 5 seconds, and the last station at 7 seconds, as shown (bits two and three are not described in text, but are shown below):

```
> (define bit-one   (list 1 3 5 7))
> (define bit-two   (list 4 6 8 10))
> (define bit-three (list 5 6 10 12))
> (define bit-list  (list bit-one bit-two bit-three))
> (latency bit-list)
7
```

The input list will never be empty. Each individual bit arrival time list will never be empty and will consist of strictly-increasing numbers. All bit arrival time lists will have equal lengths. You may always use any procedure defined in class or the problem sets.

```
(define (latency input)
     (car (reverse (car input))))
or
     (car (sort (car input) <))
or
     (max (car input))
```

In class we defined the latency of a communication instance to be the time at which **the first bit reaches the destination**. Once you understood the definition, you had to write scheme code to extract the last element from the first list in the input.
Blank: +1.5/5
subtracts last time from first time: 0 points off
English Description of Latency: +2/5
English Psuedocode: +2/5
other incorrect code: 1/5

1b. Define a `bandwidth` procedure that takes one input, a list of lists of bit arrival times (exactly as above), and produces as output a number representing the total overall bandwidth of the entire connection. Continuing the example above:

```
> (bandwidth bit-list)
1/4
```

```
(define (bandwidth input)
     (/   (length input)
          (car (reverse (car (reverse input)))) ;; last time
```

In class we defined bandwidth to be bits per second -- the total number of bits transmitted divided by the total time taken. Here the total time taken is the time at which the last bit reaches the final destiation. So you need to find two numbers:

(1) the number of bits, which is just the length of the input bit list

and

(2) the final arrival time, which is the last time in the last input list

Blank: +1.5/5
subtracts last time from first time: 0 points off
English Description of Bandwidth: +2/5
English Psueodcode: +2/5
other incorrect code: 1/5

Mutation and Databases

2a. Consider the following definition for a while loop, as per Chapter 10.4:

```
> (define (while test body)
      (if (test)
            (begin (body) (while test body))
            (void)))    ;;; no return value
```

Define a procedure `list-map!` that takes two parameters: a worker function and a list. Your `list-map!` procedure should modify the list in place, replacing each element with the result of applying the worker function to it. Your `list-map!` procedure must use `while`, and may not be directly recursive (i.e., all recursion must happen through a single call to `while`).

```
> (define x 2)
> (while    (lambda () (> x 0))
            (lambda () (display x) (set! x (- x 1))))
2 1
> x
0
> (define mylist (list 1 2 3 4))
> (list-map! (lambda (x) (* x x)) mylist)
> mylist
(1 4 9 16)
```

```
(define (list-map! workfun lst)
     (while    (lambda () (not (null? lst)))
               (lambda ()
                     (set-car! lst (workfun (car lst))
                     (set! lst (cdr lst)))))
```

The basic idea here was to use "while", instead of recursion, to iterate through a list. The while guard, just like the recursive base case, is "while not empty". The while body consists of applying the workfun to the current element and replacing the current element with the result ... and then walking down the list, by setting lst to (cdr lst).

Blank: 2.3
"while" keyword: +1
correct while guard: +1
while body calls workfun on (car lst): +1
while body calls set! twice: +2
presence of "(cdr lst)": +1
full correctness: 7/7
correct full English Psueodcode: 5/7

2b. Consider the following database table named `CSBooks`. Give a SQL `select` query that returns all book titles written in the 1990s that cost less than 14.00, sorted by author name. Be as generic as possible: support tables with different entries than the example one.

BookID	Title	Author	Price	Publisher	Year
1	The Mind's I	Hofstadter	18.95	Bantam	1985
2	GEB	Hofstadter	19.95	Basic	1979
3	Cryptonomicon	Stephenson	14.25	Perennial	1999
4	The Code Book	Singh	14.00	Anchor	2000
5	Snow Crash	Stephenson	13.75	Bantam	1992

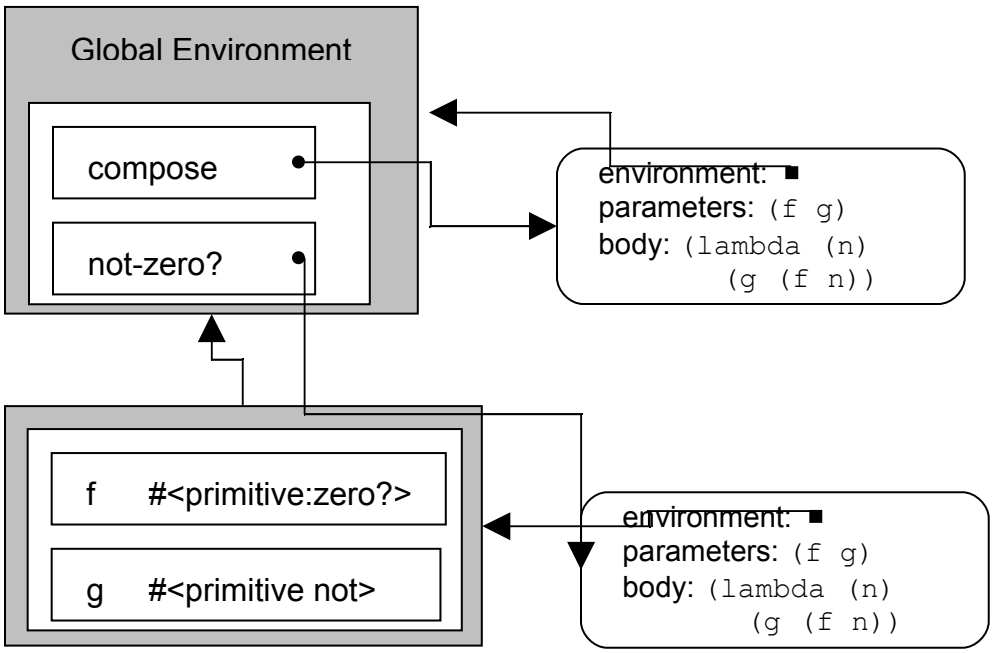
```
SELECT Title from CSBooks where Year >= 1990
     and Year < 2000 and Price < 14.00 Ordered By Author
```

A common mistake here was to do "SELECT *" instead of "SELECT Title" -- the question asks you to return the title only.

3 points total.
-1 per missing or incorrect clause.
No points off for little errors.
Blank: 1 point

Global Environments

3. Consider the environment shown below (as in question 1, assume all the usual primitives are defined in the global environment, but not shown; the notation #<primitive:zero?> denotes the primitive procedure zero?):



Provide a sequence of Scheme expressions such that evaluating the sequence of expressions produces the environment shown above. Hint: give two definitions.

This was unintentionally tricky. For what is shown, a correct answer would be:

```
(define compose (lambda (f g) (lambda (n) (g (f n)))))
(define not-zero?
   (let ((f zero?)
         (g not))
     (lambda (f g)
        (lambda (n) (g (f n))))))
```

This is not a very sensible procedure though! What the `not-zero?` procedure should have been is:

environment: (to f, g environment as before)
parameters: `(n)`
body: `(g (f n))`

Then, the answer is the more reasonable:

```
(define compose (lambda (f g) (lambda (n) (g (f n)))))
(define not-zero? (compose zero? not))
```

The top answer was worth +5 extra credit points.
Blank: +3/10
The (compose zero? not) vs. (compose not zero?) did not matter.
Having "(define compose" was +1 point.
Having "(define not-zero?" was +1 point.
Having the correct compose body was +3 points.
Having the correct not-zero? body was +5 points.
Have extra definitions was -1 points.

Object-Oriented Programming

4. Consider the following PS6-style definitions:

```
(define (make-object name1)
  (lambda (message1)
    (case message1
      ((class) 'object)
      ((name) name1)
      (else no-method))))

(define (make-person name2)
  (let ((super (make-object name2))
        (possessions '())
        (restlessness 0.0))
    (lambda (message2)
      (case message2
        ((class) 'person)
        (else (super message2))))))
```

Draw the global environment while executing:

```
(define turing (make-person 'turing))
(turing 'name)
```

Hint: be sure to draw all frames that might be created as a result of function evaluations.

It is difficult to visually represent a perfect answer for this question. First, no points were taken off if your global environment did not include make-object or make-person.

It is worth noting that there are **three** function calls in play:

(make-person 'turing)
(make-object name2)
(super message2)

The first, (make-person 'turing), creates the person object that is bound to the name turing in the global environment. The other two function calls occur when the 'name message is sent to the turing object.

Grading included one point for state variables in turing (e.g., possessions), one point for the parameter/body of make-person, one point for the parameter/body definition for make-object, one point for "message2: 'name", one point for "message1: 'name", one point for "name1: 'turing", and up to four points for correct arrows.

Computability

5. Is the *Contains-Cross-Site-Scripting-Vulnerability Problem* described below computable or uncomputable? Your answer should include a convincing argument why it is correct.

Input: *P*, a specification (all the code and html files) for a dynamic web application.

Output: If *P* contains a cross-site-scripting vulnerability, output **True**. Otherwise, output **False**.

As demonstrated in class, a cross-site-scripting vulnerability is an opportunity an attacker can exploit to get their own script running on a web page generated by the web application.

The *CSSV Problem* is **uncomputable**. We show this by arguing that if we have an algorithm, contains-css?, that solves the CSSV Problem, we could use it to solve the Halting Problem. Since we know the Halting Problem is uncomputable, this is a convincing argument that the CSSV Problem is uncomputable.

Here's how:

```
(define (halts? P)
   (contains-css?
      '(lambda ()
          (apply-procedure (remove-vulnerabilities P))
          (vulnerable-procedure))))
```

Where `vulnerable-procedure` is a procedure that is vulnerable to cross-site-scripting attacks, and `remove-vulnerabilities` is a procedure that takes a procedure specification as input, and replaces all output with empty web pages (this could be done by replacing all the print commands with something that just ignores the parameters).

Note that this is very similar to the proof in class we saw for the *Is-Virus Problem*.

4 points for saying uncomputable.
4 points for saying "if we could solve it, we could solve the Halting Problem".
2 points for the actual Scheme code of the reduction.
Blank: +3/10. Wrong answer: +2/10.

6. Is the *Remove-Cross-Site-Scripting-Vulnerabilities Problem* described below computable or uncomputable? Your answer should include a convincing argument why it is correct.

Input: *P*, a specification (all the code and html files) for a dynamic web application.

Output: *P′*, a specification for a dynamic web application. On inputs that are not cross-site-scripting attacks, *P′* behaves identically to *P*. On inputs that are cross-site-scripting attacks, *P′* ignores the attack input and displays a warning page.

For this problem, both arguments are plausible depending on how a cross-site-scripting attack is defined. First, note that the `remove-vulnerabilities` procedure we used in question 3 does not solve the *Remove-CSS Problem*. This is because the program it outputs does not behave identically to the input program on non-attack inputs. Second, we note that it is *not* necessary to be able to locate all vulnerabilities in P to solve the *Remove-CSS* problem. It is enough to transform the program dynamically, so if it happens to run on an attack input it will display the warning page. So, instead of detecting vulnerabilities, all we need to do is detect successful attacks. Hence, we can solve the Remove-CSS problem if it is possible to examine an output web page and determine if it has a script generated by a user on it (this is the definition from question 3, "a cross-site-scripting vulnerability is an opportunity an attacker can exploit to get their own script running on a web page generated by the web application".

If we can track all data through the application to know if it came from a user, then we can do this, by examining the output of the program to see if any of the data in the output that was generated from user data contains a script. It is possible to do this with an algorithm: just look for the <script …> tags (and a few other things). To track the data, we need to modify the interpreter to evaluate P, but keep extra information for each data object to indicate whether or not it came from an untrusted source. (For an example of a program that does this, see www.phprevent.org.)

If you interpreted a CSS attack more strictly to require not only getting script on the output page, but that script doing something malicious, then the problem is uncomputable. This is because it is uncomputable to determine if a script does something malicious (similarly to the Is-Virus Problem).

An Answer: +5/10.
"Run-time monitoring" or "look over shoulder": +5
You said impossible, but did not say why clearly: -3
Blank: 3/10.

Interpreters and Asymptotic Running Time

7a. For the next two questions, you are given a procedure definition. Your answer should describe its asymptotic running time when evaluated using (a) Charme, and (b) MemoCharme (the language defined at the end of PS7), and (c) LazyCharme. You may assume the Python dictionary type provides lookups with running time in O(1). Your answers should include a clear supporting argument, and define all variables you use in your answer.

```
(define duplicitous
    (lambda (a)
        (cond
            ((> a 0) (+ (duplicitous (- a 1))
                        (duplicitous (- a 1))))
            ((zero? a) 1))))
```

(a) Running time in Charme:

$\Theta(2^n)$ **[1pt]** where n is the value of a. Assuming the input is a positive integer, each evaluation of duplicitous involves *two* recursive calls **[1pt]** with the input value reduced by one. This means increasing the input value by one *doubles* the amount of work, so it scales *exponentially* in the input value.

(b) Running time in MemoCharme:

$\Theta(n)$ **[1pt]** where n is the value of a. There are still two recursive calls, but the value of whichever one is evaluated second is already memoized **[1pt]**, so the second call requires only constant time. Evaluating the first call is $\Theta(n)$ since it requires n calls to reach the base case.

(c) Running time in LazyCharme:

$\Theta(2^n)$ **[1pt]** where n is the value of a. Lazy evaluation does not help here since everything that is evaluated eagerly must also be evaluated lazily. Although lazy evaluation saves computed results, it saves them for only the particular expression that is evaluated, so it doesn't matter if there are identical expressions to be evaluated elsewhere. This is different from memoization, where the results of applying a given function to particular argument values are saved.

7b.

```
(define temeritous
    (lambda (a b)
        (cond
            ((> a 0) (temeritous (- a 1)
                                 (temeritous (+ a 1) b)))
            ((zero? a) 2))))
```

(a) Running time in Charme:

Infinite [1pt]. An application of temeritous involves a recursive call with the first parameter (+ a 1). This means the value increases with every call, moving away from the base case value of 0. Hence, evaluation never terminates and the running time is infinite.

(b) Running time in MemoCharme:

Infinite 1[pt]. Memoization provides no help here, since the input values keep increasing. We are not reusing input values, so no results will be memorized.

(c) Running time in LazyCharme:

Θ(*n*) where *n* is the value of *a* [1pt]. Here, lazy evaluation is indeed temeritous (which Wikitionary defines as "Displaying disdain or contempt for danger"). **Since the second input to temeritous is never used, it is never evaluated with lazy evaluation. [2pts]** The first input is used, and decreases by one with each recursive call, so the running time is linear in the value of the first input.

Static Type Checking

8. (as promised, Exercise 14.2) Define the `typeConditional(expr, env)` procedure that checks the type of a conditional expression. It should check that all of the predicate expressions evaluate to a Boolean value. In order for a conditional expression to be type correct, the consequent expressions of each clause produce values of the same type. The type of a conditional expression is the type of all of the consequent expressions. (You may assume the StaticCharme interpreter described in Chapter 14.)

Here is a definition to typeConditional. It is based on evalConditional, but instead of using meval, we need to use typecheck. In addition, we need to check the type of the predicate and consequence of every clause.

```
def typeConditional(expr, env):
    assert isConditional(expr)
    if len(expr) <= 2:
        evalError ("Bad conditional expression: %s" % str(expr))
    firstclause = True
    for clause in expr[1:]: // 2 pts
        if len(clause) != 2:
            evalError ("Bad conditional clause: %s" % \
                       str(clause))
        predicate = clause[0]    // 1 pt
        if not typecheck(predicate, env).matches( \
                        CPrimitiveType('Boolean')):
                // 2 pts for typecheck(...)
            return CErrorType ("Non-Boolean predicate: " + \
                               str(predicate))
        consequent = clause[1] // 1pt
        if firstclause:
            clausetype = typecheck(consequent, env)
            firstclause = False
                // 2 pts for "first clause" processing
        else:
            if not typecheck(consequent, env).matches( \
                                                clausetype):
                // 2 pts for second typecheck(...)
                return CErrorType ("Mistyped consequent: " + \
                                   str(consequent))
    return clausetype

2 pts: iterating over clauses
1 pt:  identify predictate
1 pt:  identify consequent (name does not matter)
2 pts: predicate type checks to Boolean
2 pts: consequent typchecks to Something
2 pts: all consequents have same type
Full English Pseudocode:  7/10
Blank: 3/10
```

9. (based on Exercise 14.3) A stronger type checker would require that at least one of the conditional predicates must evaluate to a true value. Otherwise, the conditional expression does not have the required type (instead, it produces a run-time error). Either define a `typeConditional` procedure that implements this stronger typing rule, or explain convincingly why it is impossible to do so.

This is impossible, assuming we want our typecheck procedure to always terminate. We show the At Least One True problem needed for the stronger type checker is undecidable by showing that an algorithm that solves it could be used to solve the Halting Problem. Here's how:

```
(define (halts? P)
    (at-least-one-true?
      '(cond ((begin (apply-procedure P) #t) 0))))
```

If P halts, then the (only) clause predicate evaluates to true, and at-least-one-true? would evaluate to true. If P does not halt, the predicate would never finish evaluating, so no clause evaluates to true.

Note that this result does not mean it isn't worth trying to solve this problem. Indeed, many program verifiers do this. There are programs that attempt to prove correctness properties of other programs. They attempt to prove that at least one of the conditional expressions evaluates to true. In some cases this is easy (for example, when the last predicate is the literal expression #t), sometimes it is possible (for example, when the first predicate is (< a 0) and the second predicate is (>= a 0)), and sometimes it is impossible. For a program verifier to be useful, it should always terminate, but sometimes it will not be able to verify a correct program.

Note that is it not sensible to just replace the typecheck with meval to attempt to find if a predicate evaluates to true. The values of parameters are not known when a definition is type checked, so there is no way to evaluate the predicate expressions (if they involve values that are not yet known).

Saying "undecidable": 4 points
Saying "because it could be used to solve the halting problem": 4 points
Correct Scheme code for the reduction: 2 points
Huge code that tries to do the typechecking: 4 points
Pseudocode that tries to do the typechecking: 3 points
Blank: 3/10

End of Exam