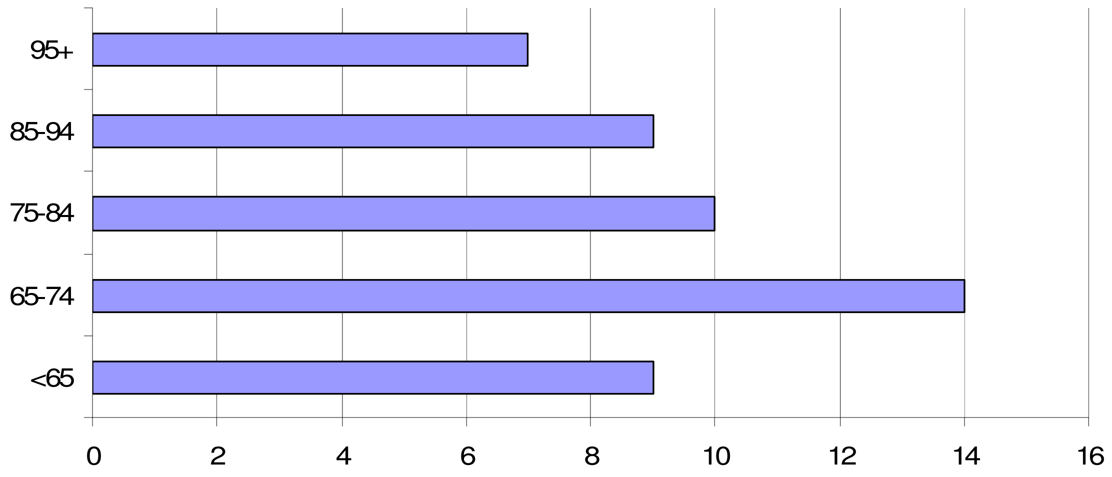
cs150: Exam 2 Comments

Scores

Question Averages:

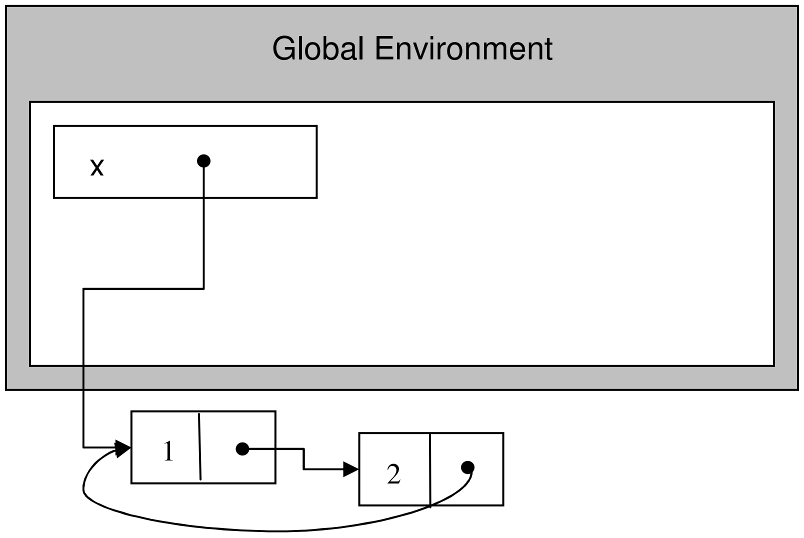
0	1	2	3	4	5	6	7	8	9	Total
9.8	7.8	8.5	8.2	6.6	8.5	7.1	7.4	6.3	6.7	75.4
10	10	10	10	10	10	10	10	10	10	100

Histogram



Environments

1. Consider the environment shown below (assume all the usual primitives are defined in the global environment, but not shown):

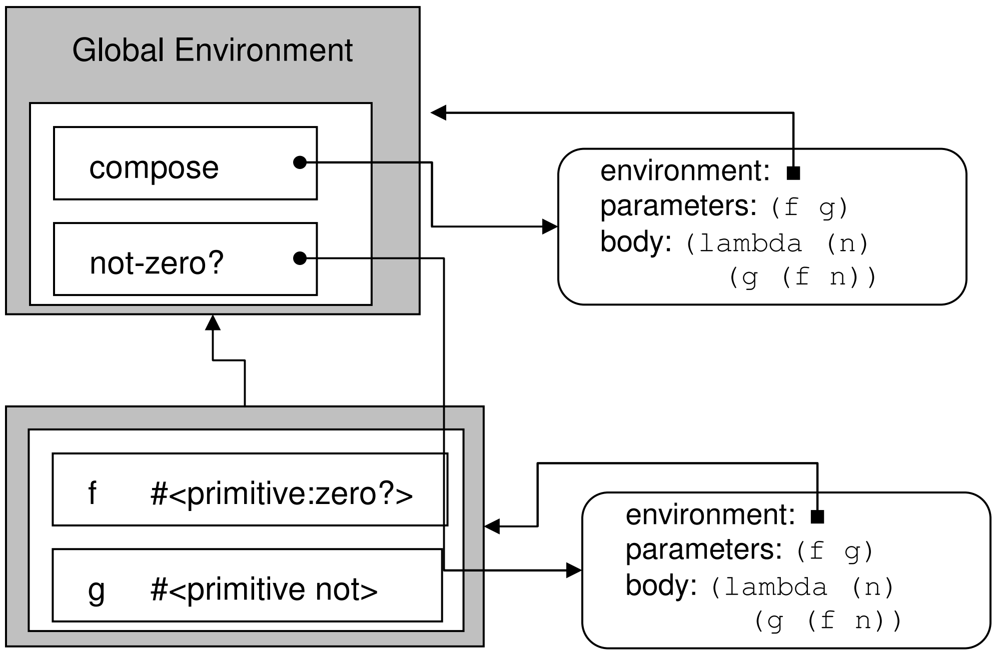


Provide one additional expression that follows the given definition, such that the environment shown above results when the sequence of expressions is evaluated.

```
(define x (cons 1 (cons 2 3)))
```

```
(set-cdr! (cdr x) x)
```

2. 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?):

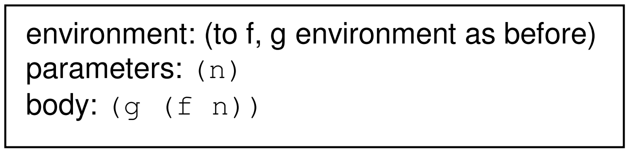


Provide a sequence of Scheme expressions such that evaluating the sequence of expressions produces the environment shown above.

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:



Then, the answer is the more reasonable:

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

(Full credit was given for either answer, or any answer that reflected the new environment correctly.)

Computability

3. 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. 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).

Interpreters and Asymptotic Running Time

For the next three 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.

5.
```
(define mysterious
    (lambda (a)
       (cond
          ((> a 0) (mysterious (- a 1)))
          ((zero? a) 0))))
```

(a) Running time in Charme:

$\Theta(n)$ where n is the value of a. Assuming the input is a positive integer, there will be n recursive calls (until it reaches 0), and the work for each call is constant time.

(It is important to note that it is the *value* not the *size* of a. The running time is $\Theta(2^s)$ where s is the size of a, since the maximum value that can be represented in s bits is 2^s.)

(b) Running time in MemoCharme:

$\Theta(n)$. Memoization provides no help here, since the input value is different for each call. In the case where there had been previous evaluations, though, if mysterious is re-applied to an input value used previously, the running time is constant, O(1).

(c) Running time in LazyCharme:

$\Theta(n)$. Lazy evaluation provides not difference here, since everything that is evaluated eagerly is also needed and must be evaluated lazily

6.

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

(a) Running time in Charme:

$\Theta(2^n)$ where n is the value of a. Assuming the input is a positive integer, each evaluation of duplicitous involves *two* recursive calls 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:

$\Theta(n)$ 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, 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:

$\Theta(2^n)$ 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.

7.

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

(a) Running time in Charme:

Infinite. 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. 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:

$\Theta(n)$ where n is the value of a. 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. 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 14.1) 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:]:
        if len(clause) != 2:
            evalError ("Bad conditional clause: %s" % \
                       str(clause))
        predicate = clause[0]
        if not typecheck(predicate, env).matches( \
                     CPrimitiveType('Boolean')):
            return CErrorType ("Non-Boolean predicate: " + \
                               str(predicate))
        consequent = clause[1]
        if firstclause:
            clausetype = typecheck(consequent, env)
            firstclause = False
        else:
            if not typecheck(consequent, env).matches( \
                                              clausetype):
                return CErrorType ("Mistyped consequent: " + \
                                   str(consequent))
    return clausetype
```

9. (based on Exercise 14.2) 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).