University of Virginia **Out: 5 December 2011**
cs1120: Introduction of Computing **Due: 1:00pm, Monday, 12 December 2011**
Explorations in Language, Logic, and Machines

Final Exam

Name: ________________________________	**UVa Email ID:** __ __ __ __ __ __

Directions

Due: 1:00pm, Monday, 12 December 2011. This is a hard deadline. I am leaving on a trip shortly after this, and will need to assign final grades based on what I have received by 1pm on Monday, 12 December. You may turn in your exam by giving it to me in person at my office if you find me there, or sliding it under my office door if I am not there.

Work alone. Between receiving this exam and turning it in, **you may not discuss these problems or anything directly related to this exam with anyone other than the course staff.**

Open resources. You may use any books you want, lecture notes, slides, your notes, and problem sets, including any materials posted on or linked from the course website. You may also use any computer programs you want, including DrRacket, Python, and code from the problem sets including PS7. You may also use external non-human sources including books and web sites. If you use anything other than the course book, slides, notes, and standard interpreters, you must cite what you used clearly in your answer. **You may not obtain any help from other humans** other than the course staff (who will only answer clarification questions).

Answer well. A "full credit" answer for each question is worth 10 points (but it is possible to get more than 10 points for especially elegant and insightful answers). Your answers must be clear enough for me to read and understand. Write your answers in the spaces provided after each question. You should not need more space than is provided to write good answers, but if you need more you may use the backs or attach extra sheets. If you do, make sure the answers are clearly marked.

The questions are not necessarily in order of increasing difficulty. There is no time limit on this exam, but it should not take a well-prepared student more than two hours to complete.

Full credit depends on the clarity and elegance of your answer, not just correctness. Your answers should be as short and simple as possible, but not simpler. Your answers will be judged for correctness, clarity and elegance, but you will not lose points for trivial errors (such as missing a closing parenthesis).

Pledge: ____
Sign here to indicate that you **read, agreed to, and followed** all of the directions here in addition to the Course Pledge.

Language

1. Write a small BNF replacement grammar that produces *exactly* 27 strings. For full credit, your grammar should use only two different non-terminals and no more than three terminals.

2. Write a BNF replacement grammar that produces all strings made up of **0**s and **1**s that end in a **0** and no other strings. For example, your grammar should produce the strings **01001010** and **0**, but not **10101** or the empty string.

Defining Procedures

A famous unsolved problem in mathematics is the Collatz conjecture:

1. Start with any natural number, *n*.
2. If *n* is even, divide *n* by two. If *n* is odd, multiply it by 3 and add 1.
3. Keep going until you reach one.

For example, starting from $n = 1120$, we would go through this sequence:

560, 280, 140, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

The Collatz conjecture speculates that no matter what number you start with, this process eventually reaches 1. It is not known if this is true (but no one has yet found a natural number for which it fails).

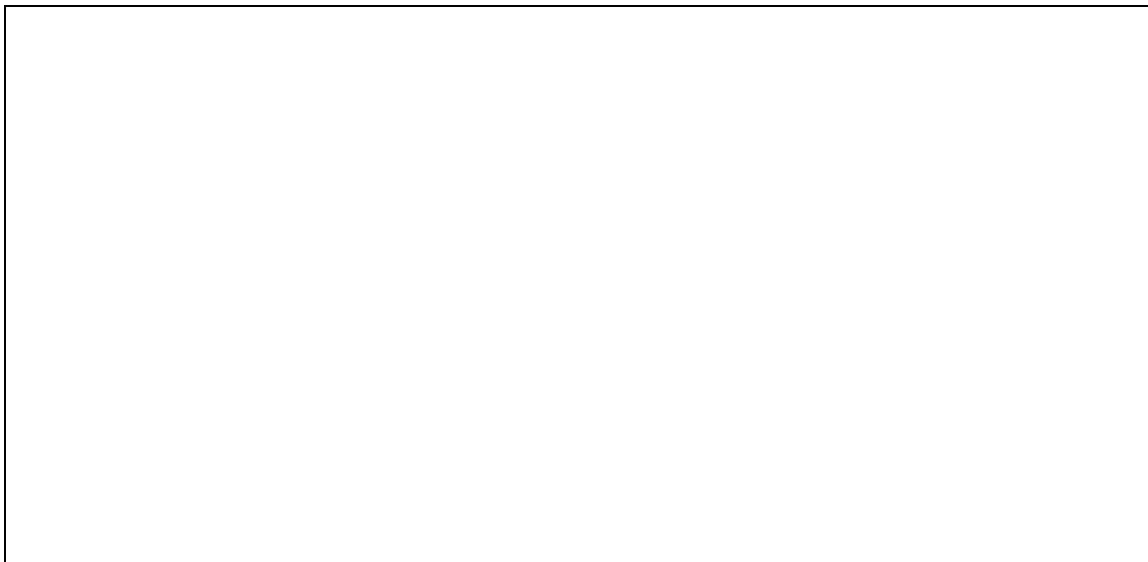
3. Define a procedure using either Scheme or Python (your choice) that takes as input a natural number *n* and tests the Collatz conjecture for *n*. Your procedure should output the sequence of values on the path to reaching one as a list. If the Collatz conjecture is false (that is, the value never reaches 1), your procedure may run forever.

For example, in Scheme:

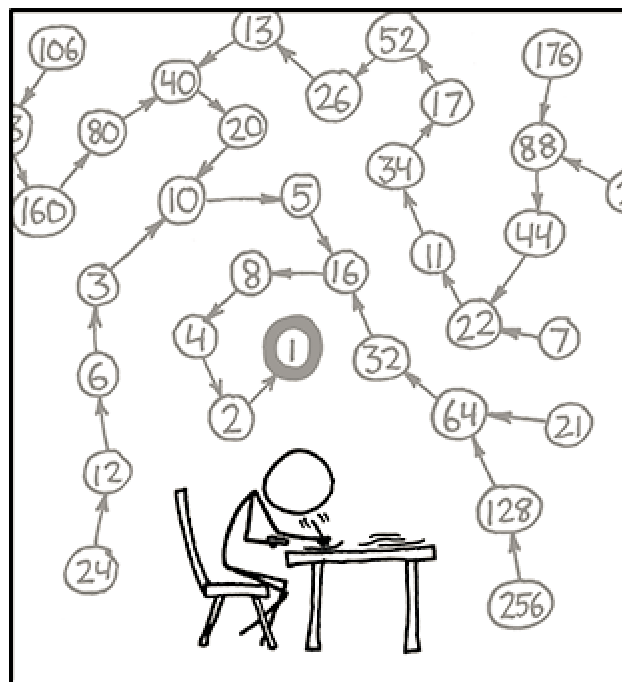
```
> (collatz 1120)
(1120 560 280 140 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1)
```

in Python:

```
>>> collatz(1120)
[1120, 560, 280, 140, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```



[Bonus] Explain why determining if a correct collatz procedure (that solves the previous question) is an *algorithm* would be worth at least a quadruple-gold-star.



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

xkcd.com

Running Time Analysis

4. For each procedure below, give the worst-case asymptotic running time. Remember to define all variables you use in your answer clearly, and state any assumptions.

   a.

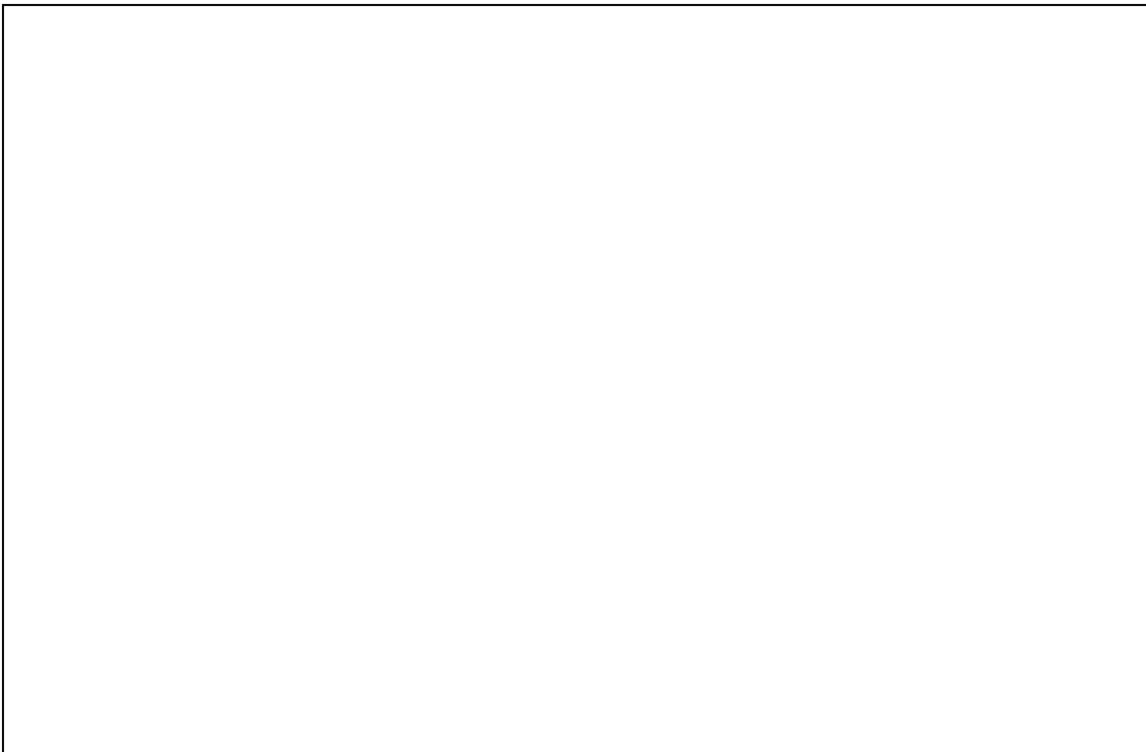
```
(define (mlist-copy p)
   (if (null? p)
      null
      (mcons (mcar p) (mlist-copy (mcdr p)))))
```

b.

```
def isSquare(n):
   for i in range(1, n):
      if i * i == n:
         return True
   return False
```

c. A magic square is a square where each row, column, and both diagonals sums to the same value. Give the asymptotic running time of this Python procedure that tests if a square passed in as a list of lists is a magic square. (You may assume all arithmetic operations used are constant time, and that the input p is a square.)

```
def isMagicSquare(s):
    target = 0
    for e in s[0]:
        target = target + e
    diagsum = 0
    revdiagsum = 0
    for i in range(0, len(s)):
        colsum = 0
        rowsum = 0
        diagsum = diagsum + s[i][i]
        revdiagsum = revdiagsum + s[i][len(s) - i - 1]
        for j in range(0, len(s)):
            rowsum = rowsum + s[j][i]
            colsum = colsum + s[i][j]
        if colsum != target or rowsum != target: return False
    return diagsum == target and revdiagsum == target
```



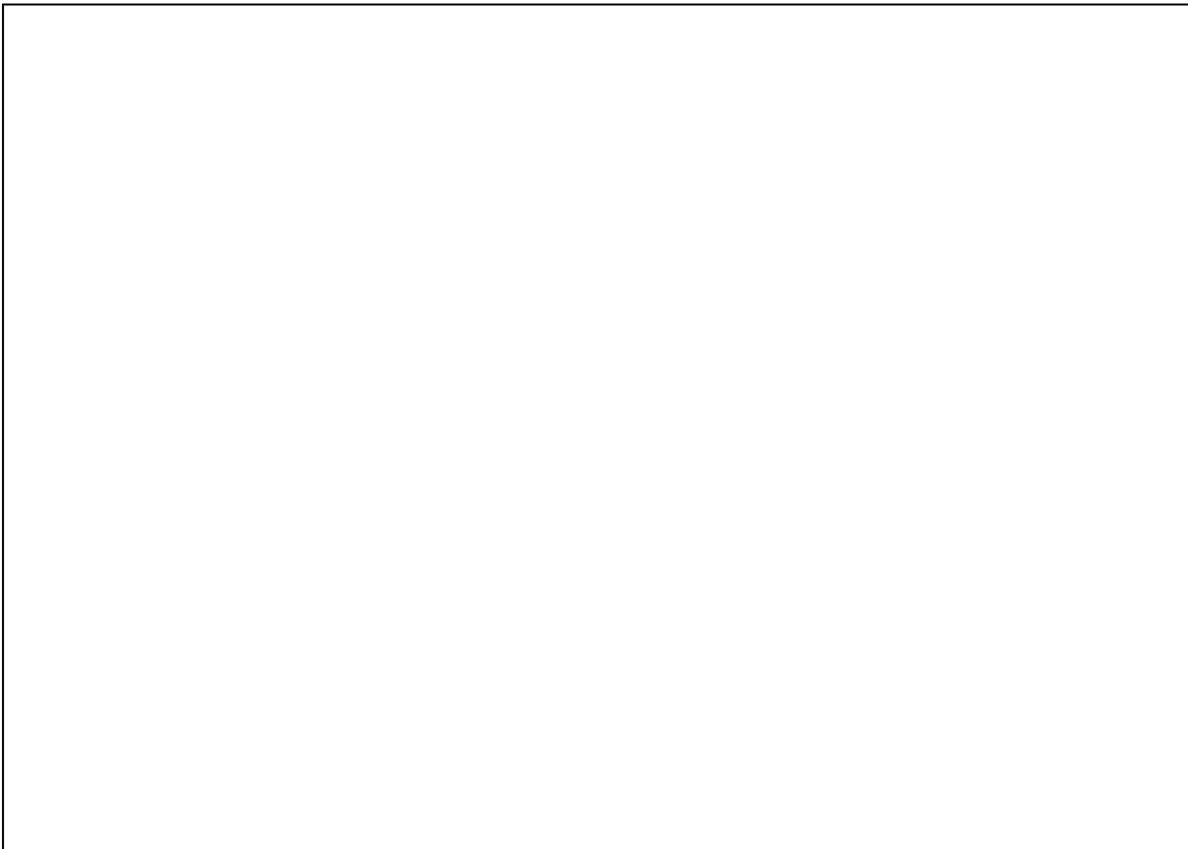
Mutation

5. Define a procedure that takes as input a mutable list of numbers, and modifies the list so that every other element is replaced with its negation. So, the first element should be negated, but not the second, etc. You may use *either* Scheme or Python to define your procedure (your choice). For example, in Scheme you would define the **mlist-negate-every-other!** procedure that behaves like this:

```
> (define p (mlist 1 -2 3 0 -17))
> (mlist-negate-every-other! p)
> p
{-1 -2 -3 0 17}
```

In Python you would define the **listNegateEveryOther** procedure that behaves like this:

```
>>> p = [1, -2, 3, 0, -17]]
>>> listNegateEveryOther(p)
>>> p
[-1, -2, -3, 0, 17]
```



6. Define a Scheme or Python procedure that takes as input a mutable list, and modifies the list so that each element is the maximum value of all elements up to and including itself. Your procedure must modify the input list.

   For example, in Scheme:

   ```
   > (define p (mlist 1 2 3 2 4 6 3))
   > (make-maxilative! p)
   > p
   {1 2 3 3 4 6 6}
   ```

   For example, in Python:

   ```
   >>> p = [1, 2, 3, 2, 4, 6, 3]
   >>> makeMaxilative(p)
   >>> p
   [1, 2, 3, 3, 4, 6, 6]
   ```

Interpreters

Suppose we want to add a new special form to Charme similar to the **for** construct in Python. The grammar for the **for** special form is:

Expression → *ForExpression*
ForExpression → **(for** *Name* **in** $Expression_1$ **do** $Expression_2$**)**

The first expression should evaluate to a list; the second expression can be any expression. The evaluation rule is:

> To evaluate the **for** expression, **(for *N* in E_1 do E_2)**, evaluate the expression E_1, which must evaluate to a list. Then, for each element in the list evaluate the expression E_2 in a new environment where the name ***N*** is bound to the value of that element. The value of the for expression is a list containing the results of each evaluation of E_2.

Here are a few examples:

```
> (for x in (list 1 2 3 4) (+ x 1))
(2 3 4 5)
> (for x in (list 2 3) (* x x))
(4 9)
```

7. Define an **evalFor(expr, env)** procedure that implements the evaluation rule for the and **for** special form as defined above. You may assume the value passed in as **expr** is parsed Charme expression and that corresponds to a syntactically valid **for** expression.

8. Would it be a good idea to add **for** to Charme (or Scheme)? Provide a convincing argument supporting your answer.

Computability

9. Is the *RUNS-FOREVER* problem defined below computable? For full credit, your answer must include a clear and convincing proof supporting your answer.

   Input: A string *s* that defines a Python program.

   Output: True if the program defined by *s* runs forever. False if the program defined by s will eventually finish.

10. Is the *RUNS-LONGER* problem defined below computable? For full credit, your answer must include a convincing proof supporting your answer.

    Input: Strings that define two Python procedures, p and q.

    Output: True if the procedure defined by p takes longer to finish running than the procedure defined by q. A procedure *takes longer to finish* than another procedure if the first procedure finishes in K steps (for some number K), and the second procedure does not finish with K steps.

Problem	Score	Notes
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
Total		